



TotalView Training

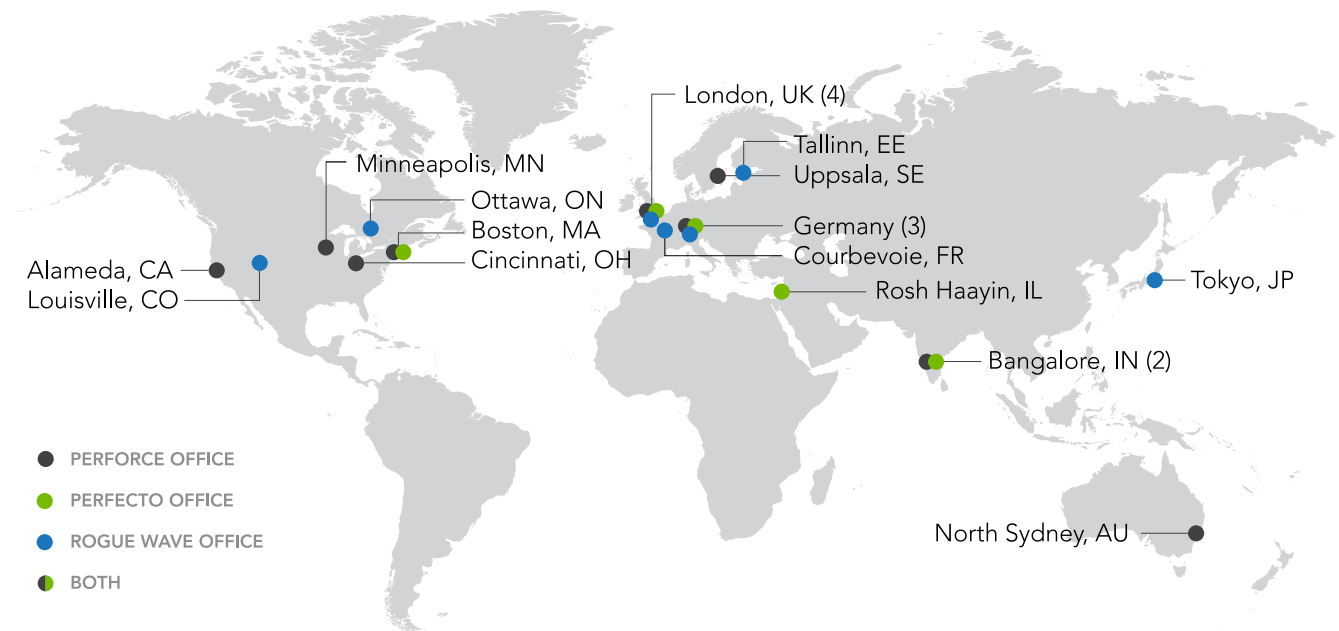
DECEMBER 2020

LBL/NERSC Agenda

- Introduction (Bill)
- Overview of TotalView Labs (Bill/Dean)
- TotalView Features (Bill)
- TotalView's New UI (Dean)
- Remote Debugging – Remote Display Client (Dean)
- Remote Debugging – Remote UI with Reverse Connect Feature (Bill)
- Startup (Dean)
- UI Navigation and Process Control (Dean)
- Action Points (Dean)
- Examining and Editing Data (Dean)
- Advanced C++ and Data Debugging (Dean)
- Q&A
- **Break**
- Python Debugging (Dean)
- Replay Engine (Dean)
- OpenMP debugging (Bill)
- MPI Debugging (Bill)
- MemoryScape (Dean or Bill)
- CUDA debugging (Bill)
- TotalView Roadmap (Bill)
- Common TotalView usage hints
- Q&A
- TotalView Labs Help

Perforce Global Footprint

- Customers in 80 countries.
- ~9,000 customers worldwide.
- More than 250 of the Fortune 500.
- Customers deploying multiple products.
- 25+ offices and 4 data centers which give us global reach.
- Over 900 employees in 25 countries.



Perforce Product Portfolio



Agile Management

Helix ALM

Hansoft

Gliffy



Code Management & Collaboration

Helix Core

Helix4Git

JRebel

TotalView



Application Mgmt. & Components

Akana

OpenLogic

Zend

Visualization

SourcePro

IMSL



Automated Testing

Helix QAC

Perfecto

Klocwork

Overview of TotalView Labs

Overview of TotalView Labs

Nine different labs and accompanying example programs

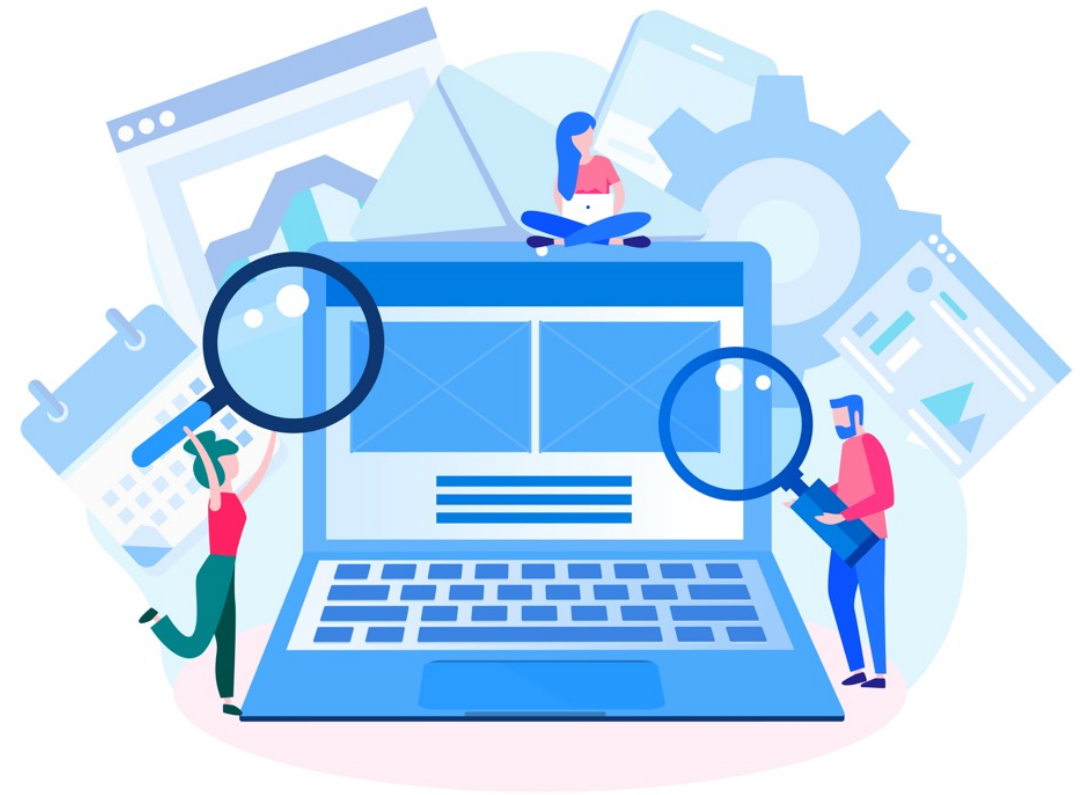
- Lab 1 - Debugger Basics: Startup, Basic Process Control, and Navigation
- Lab 2 - Viewing, Examining, Watching, and Editing Data
- Lab 3 - Examining and Controlling a Parallel Application
- Lab 4 - Exploring Heap Memory in an MPI Application
- Lab 5 - Debugging Memory Comparisons and Heap Baseline *
- Lab 6 - Memory Corruption discovery using Red Zones *
- Lab 7 - Batch Mode Debugging with TVScript
- Lab 8 - Reverse Debugging with ReplayEngine
- Lab 9 - Asynchronous Control Lab

Notes

- Labs 5 and 6 require use of TotalView's Classic UI
- Sample program breakpoint files were created with GNU compilers. If a different compiler is used, they may not load and will need to be recreated.
- Several example programs use OpenMPI so you will need to configure your environment beforehand.
- We do not have a lab specific to Python Debugging yet. There are good examples and instructions in the TotalView *totalview.2020.3.11/<linux-x86-64>/examples/PythonExamples* directory.

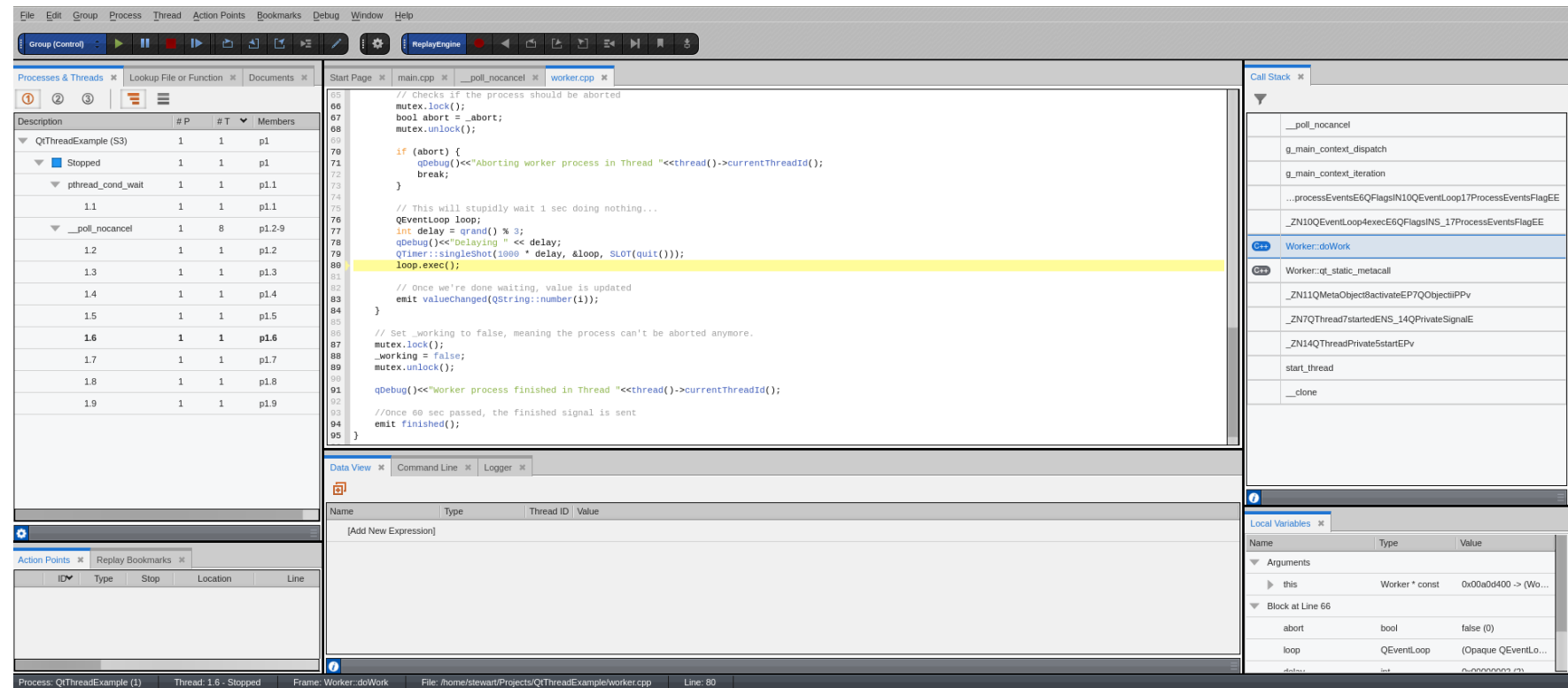
What is TotalView used for?

- Provides interactive Dynamic Analysis capabilities to help:
 - Understand complex code
 - Improve code quality
 - Collaborate with team members to resolve issues faster
 - Shorten development time
- Finds problems and bugs in applications including:
 - Program crash or incorrect behavior
 - Data issues
 - Application memory leaks and errors
 - Communication problems between processes and threads
 - CUDA application analysis and debugging
- Contains batch and Continuous Integration capabilities to:
 - Debug applications in an automated run/test environment



TotalView Features

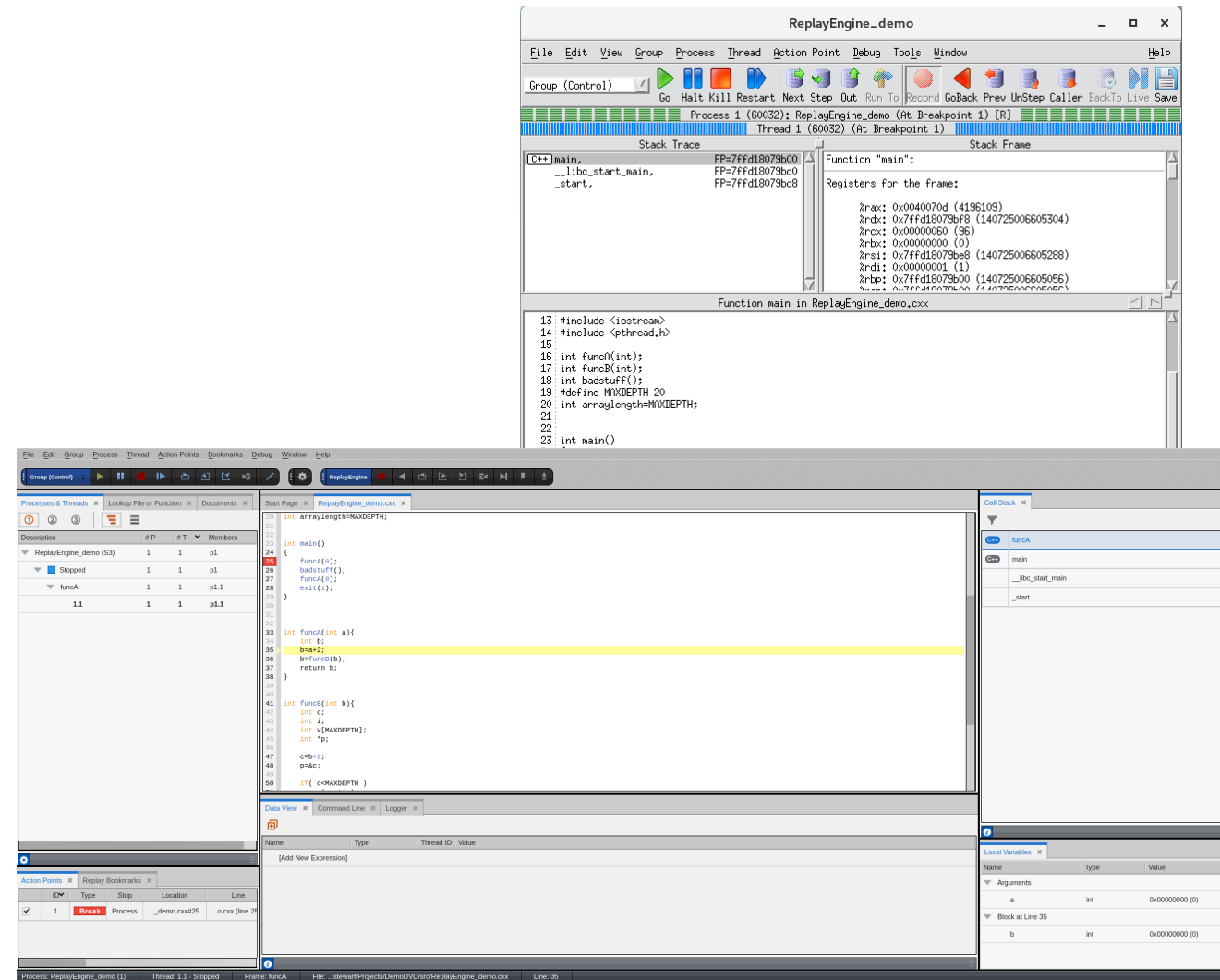
- Comprehensive C, C++ and Fortran multi-process/thread dynamic analysis and debugging
 - Thread specific breakpoints with individual thread control
 - View thread specific stack and data
 - View complex data types easily
- MPI and OpenMP HPC debugging
- CUDA debugging
- Integrated Reverse debugging
- Mixed Language - Python C/C++ debugging
- Memory leak detection
- Batch/unattended debugging
- Linux, macOS and UNIX
- **More than just a tool to find bugs**
 - Understand complex code
 - Improve developer efficiency
 - Collaborate with team members
 - Improve code quality
 - Shorten development time



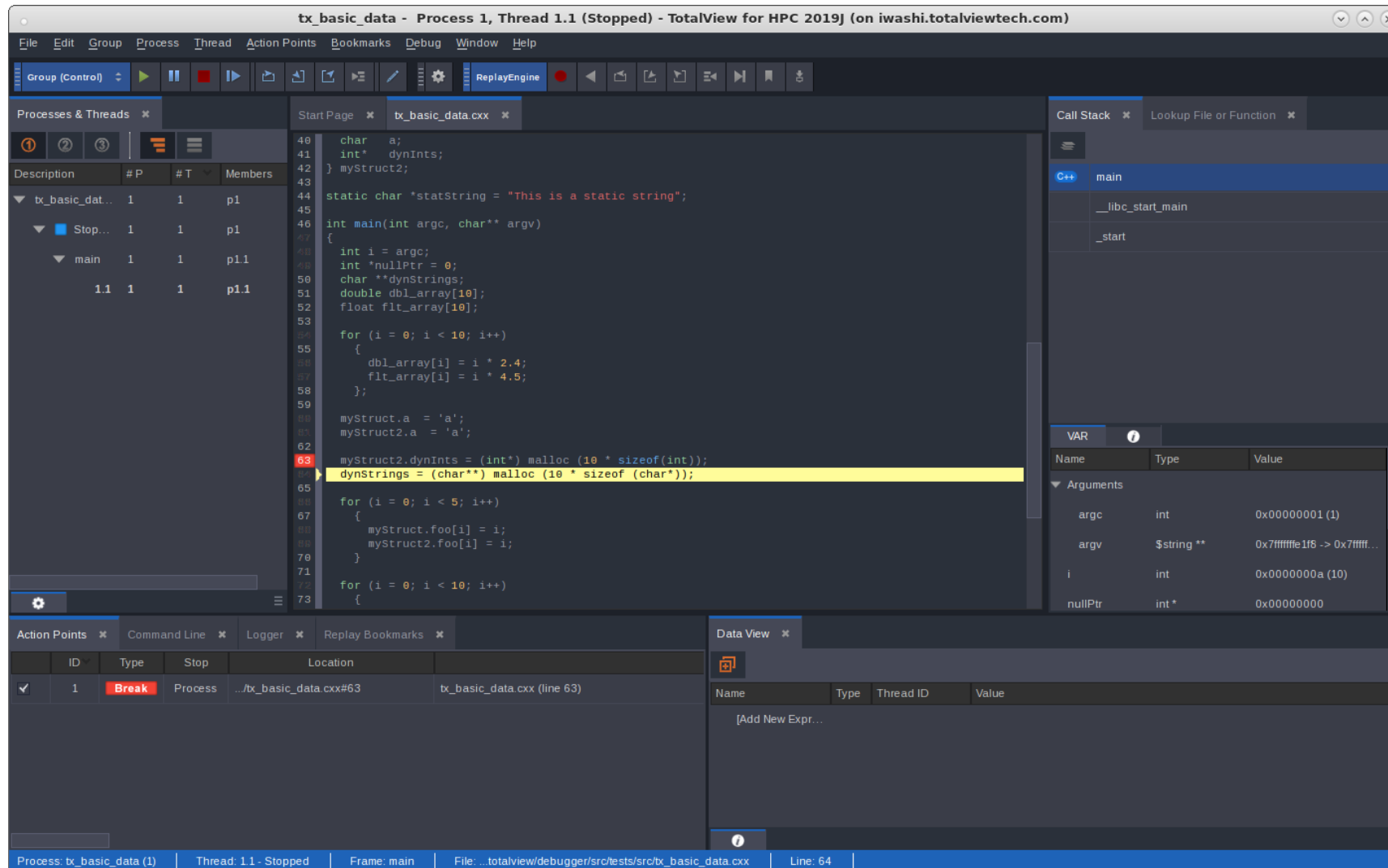
TotalView's New UI

TotalView's New UI

- Provides a modern, dockable interface
- Easier to use, better workflows
- An architecture to grow
- To use:
 - Set UI preference
 - Or command line argument
`totalview -newUI` or `totalview -oldUI`
- New UI gaps:
 - Missing array viewer, data visualization
 - Memory debugging not integrated
 - No high-scale MPI support



TotalView's Dark Theme – 2019.1



Remote Debugging

Remote Debugging Sessions

- Debugging on a remote HPC cluster can be a challenge
 - Setting up the secure remote connection
 - Launching/connecting to the target application
 - Interactive debugging UI
- TotalView Remote Debugging Options
 - **TotalView Remote Display Client**
 - Conveniently setup a remote VNC connection
 - **TotalView Reverse Connect**
 - Disconnect launching the core debugger within the cluster from the UI on a front-end node
 - **TotalView Remote UI**
 - Run the TotalView UI on a remote client and connect to the remote TotalView debugger



TotalView Remote Display Client (RDC)

- Offers users the ability to easily set up and operate a TotalView debug session that is running on another system
- Consists of three components
 - Client – runs on local machine
 - Server – runs on any system supported by TotalView and “invisibly” manages the secure connection between host and client
 - Viewer – window that appears on the client system
- Remote Display Client is available for:
 - Linux x86, x86-64
 - Windows
 - Mac OSX

TotalView Remote Display Client

- Free to install on as many clients as needed
- No license required to run the client
- Presents a local window that displays TotalView or MemoryScape running on the remote machine
- Requires SSH and X Windows on Server

TotalView Remote Display Client

- User must provide information necessary to connect to remote host
- Passwords are NOT stored
- Information required includes:
 - User name, public key file, other ssh information
 - Directory where TotalView/MemoryScape is located
 - Path and name of executable to be debugged
 - If using indirect connection with host jump, each host
 - Host name
 - Access type (User name, public key, other ssh information)
 - Access value
- Client also allows for batch submission via PBS Pro or LoadLeveler

TotalView Remote Display Client

The screenshot shows the 'TotalView Remote Display Client' window. It has a menu bar with 'File' and 'Help'. The main area is titled 'TotalView by Perforce'. On the left, there's a 'Session Profiles' sidebar with a list containing 'ntk-vnc1.totalviewtech.com'. The main area is divided into three numbered steps for configuring a debug session.

1. Enter the Remote Host to run your debug session:

Remote Host: User Name:

2. As needed, enter hosts in access order to reach the Remote Host:

	Host	Access By	Access Value	Commands
1		User Name		
2		User Name		

3. Enter settings for the debug session on the Remote Host :

Path to TotalView on Remote Host:

Arguments for TotalView:

Your Executable (path & name):

Arguments for Your Executable:

Submit Job to Batch Queueing System:

No session running

Session Profile Management

- Connection information can be saved as a profile, including all host jumping information
- Multiple profiles can be generated
- Profiles can be exported and shared
- Generated profiles can be imported for use by other users

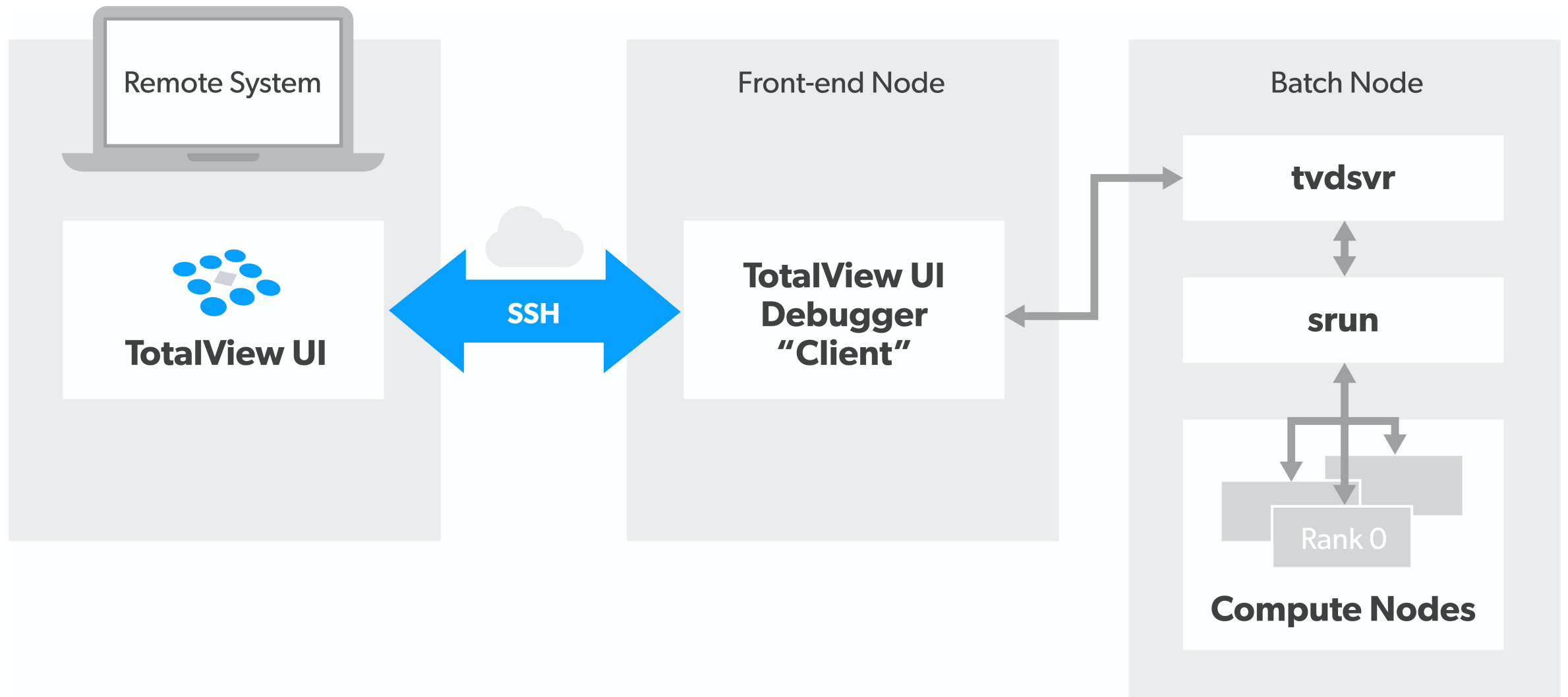
RDC Demo

- Remote Display Client demo (Linux)

TotalView Remote UI (TotalView 2020.3)

- Combine the convenience of establishing a remote connection to a cluster and the ability to run the TotalView GUI locally.
- Front-end GUI architecture does not need to match back-end target architecture (macOS front-end -> Linux back-end)
- Secure communications
- Convenient saved sessions
- Supports reverse connections

Remote UI Architecture



TotalView Reverse Connections

- The organization of modern HPC systems often makes it difficult to deploy tools such as TotalView
- The compute nodes in a cluster may not have access to any X libraries or X forwarding
- Launching a GUI on a compute node may not be possible
- Using the Reverse Connect feature you can start the TotalView UI on a front end node and when a job is run in the cluster it connects back to the waiting UI

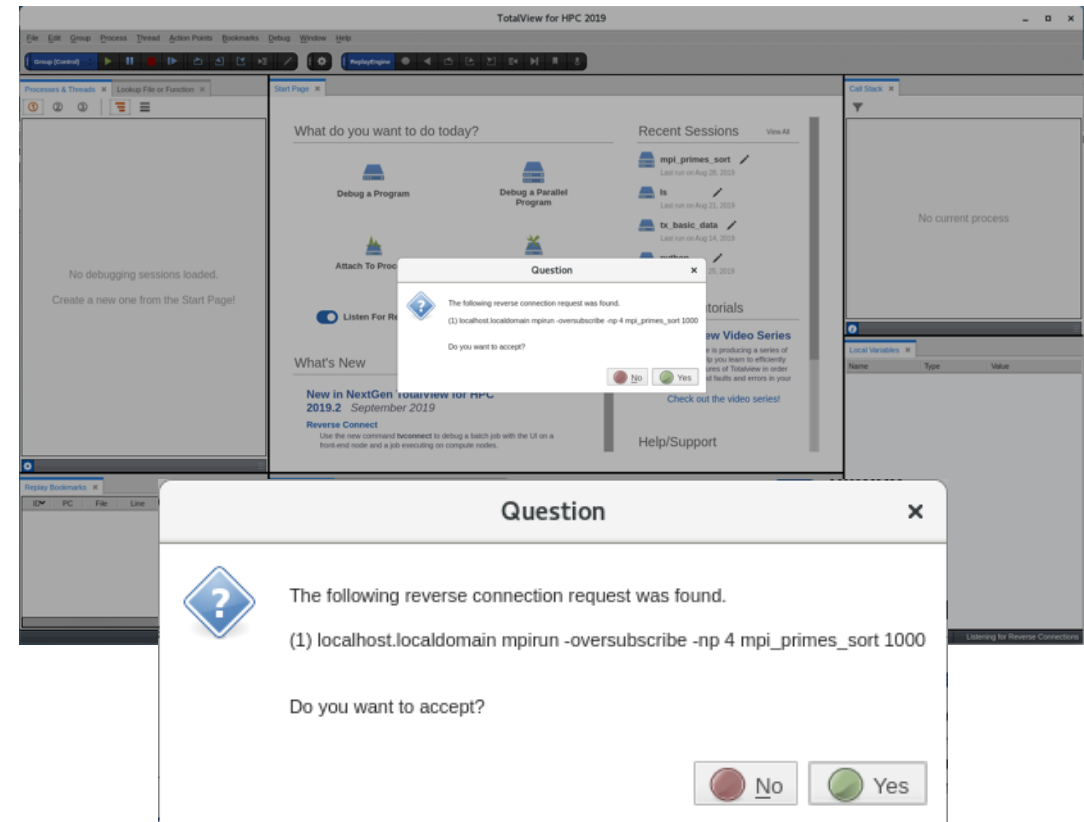
Reverse Connect

- Useful in batch environments, where
 - X11 libraries or X forwarding are not available
 - You typically use batch scripts
- TotalView's reverse connect feature
 - Disconnects starting debugger UI from backend job launch
 - Starts a TotalView server on the batch node

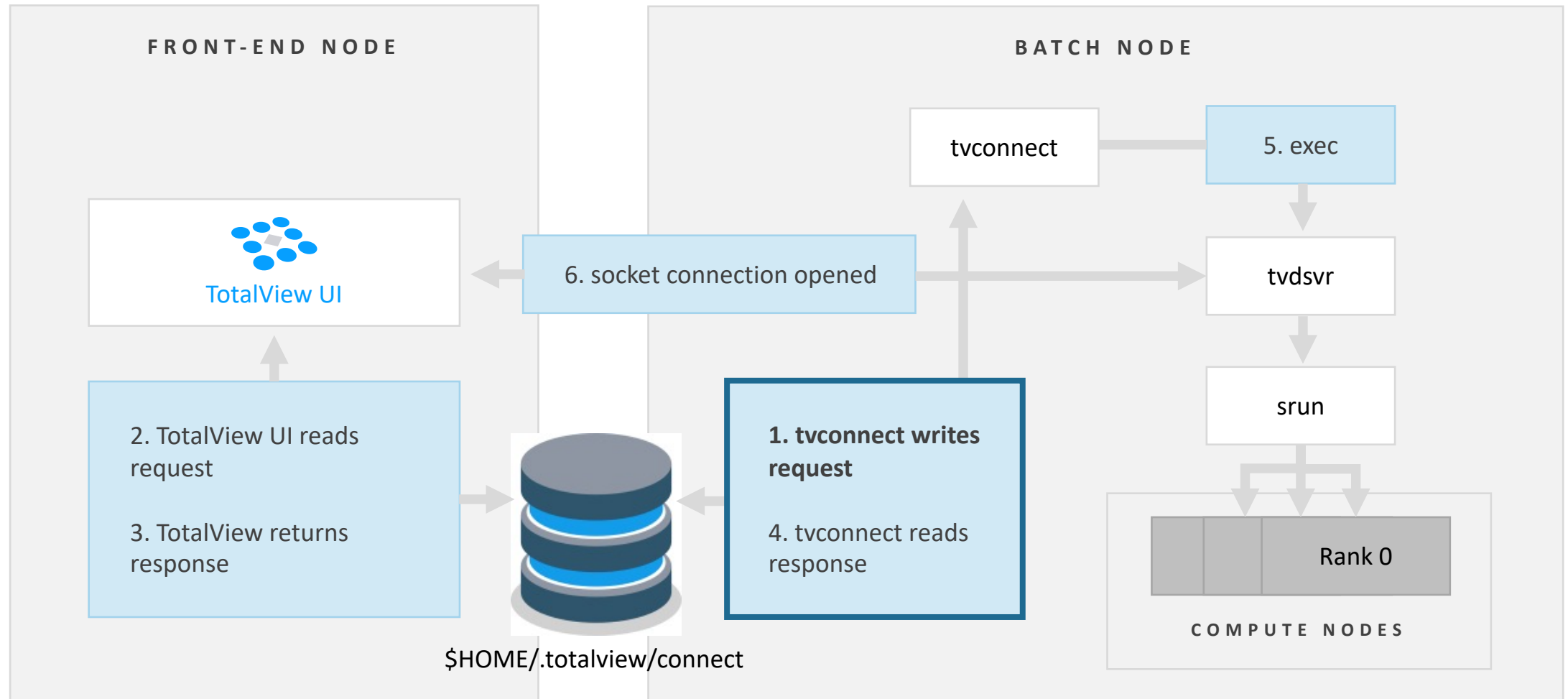
Batch Script Submission with Reverse Connect

- Start a debugging session using TotalView Reverse Connect.
- Reverse Connect enables the debugger to be submitted to a cluster and connected to the GUI once run.
- Enables running TotalView UI on the front-end node and remotely debug jobs executing on the compute nodes.
- Very easy to utilize, simply prefix job launch or application start with “tvconnect” command.

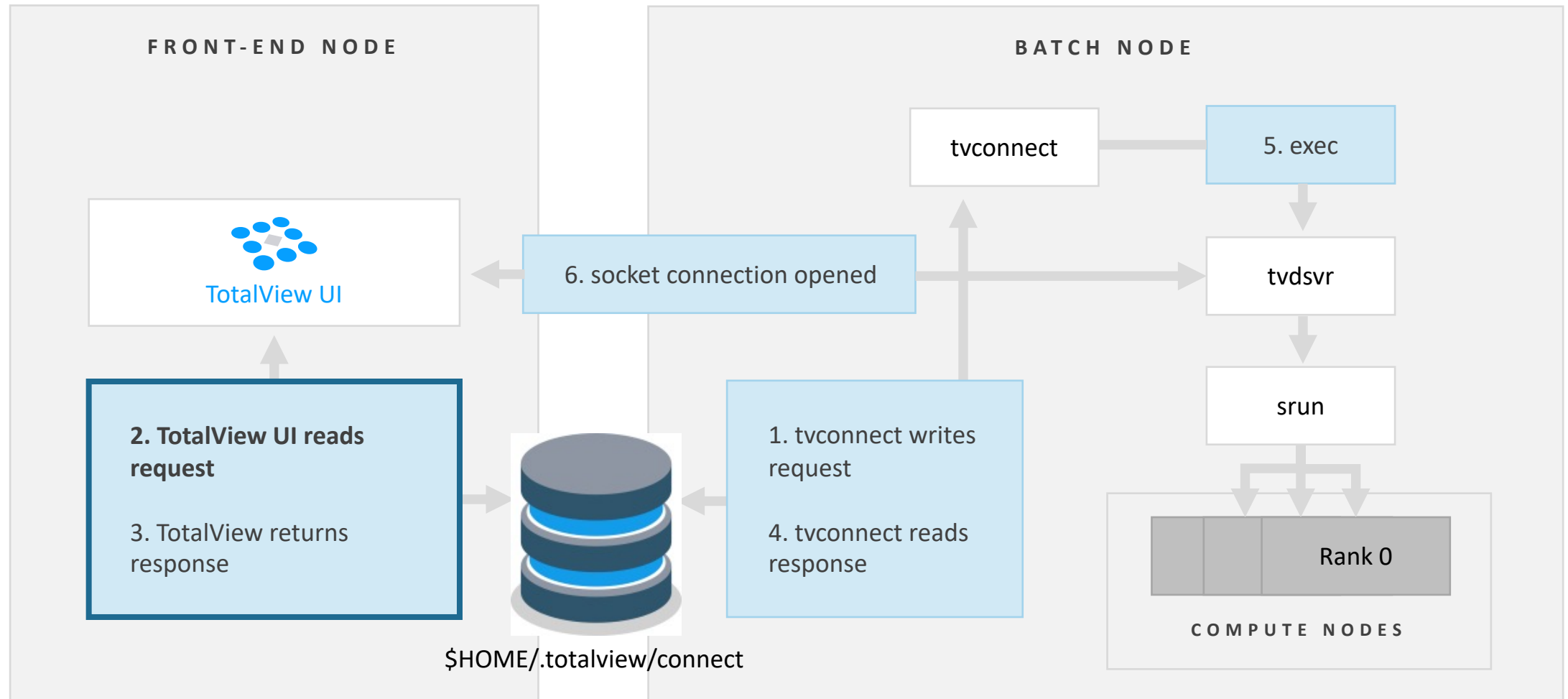
```
#!/bin/bash
#SBATCH -J hybrid_fib
...
#SBATCH -n 2
#SBATCH -c 4
#SBATCH --mem-per-cpu=4000
export OMP_NUM_THREADS=4
tvconnect srun -n 2 --cpus-per-task=4 --mpi=pmix ./hybrid_fib
```



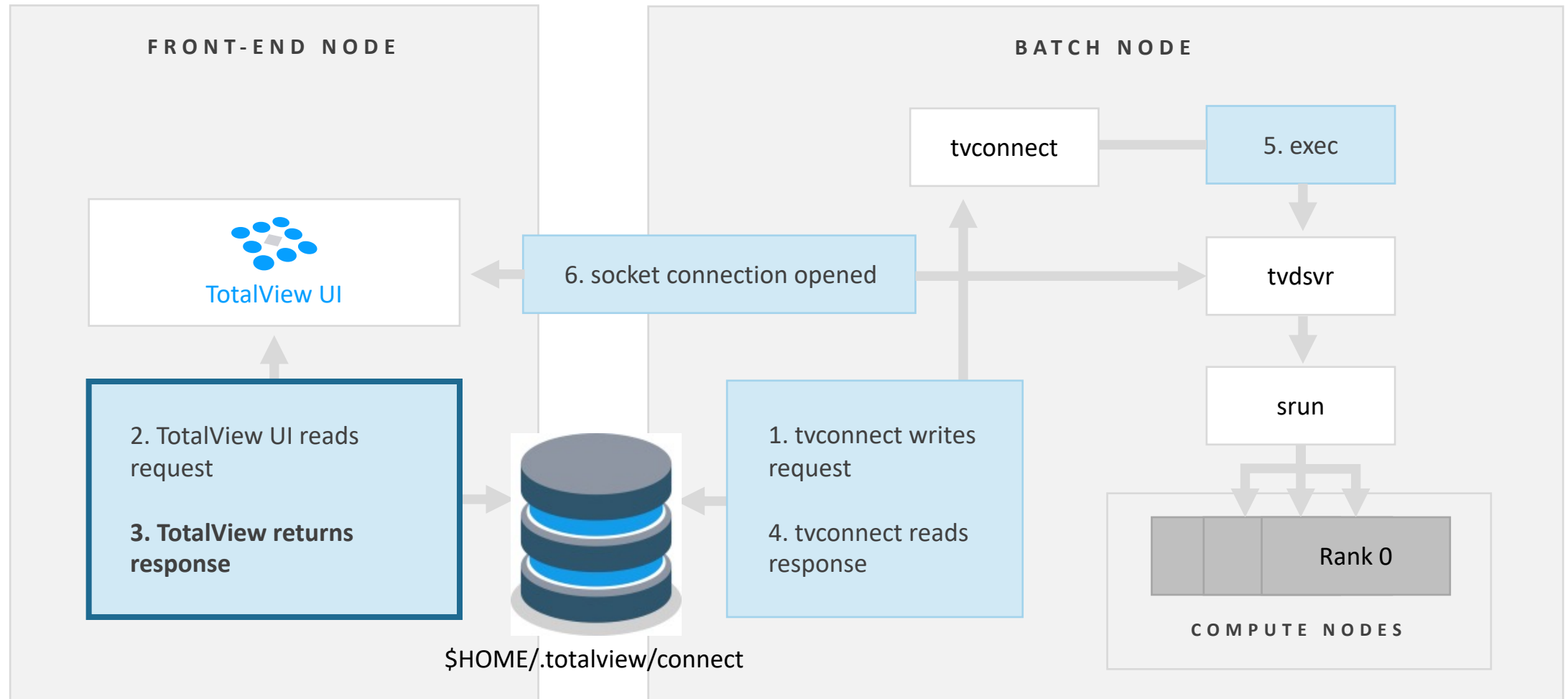
Reverse Connection Flow



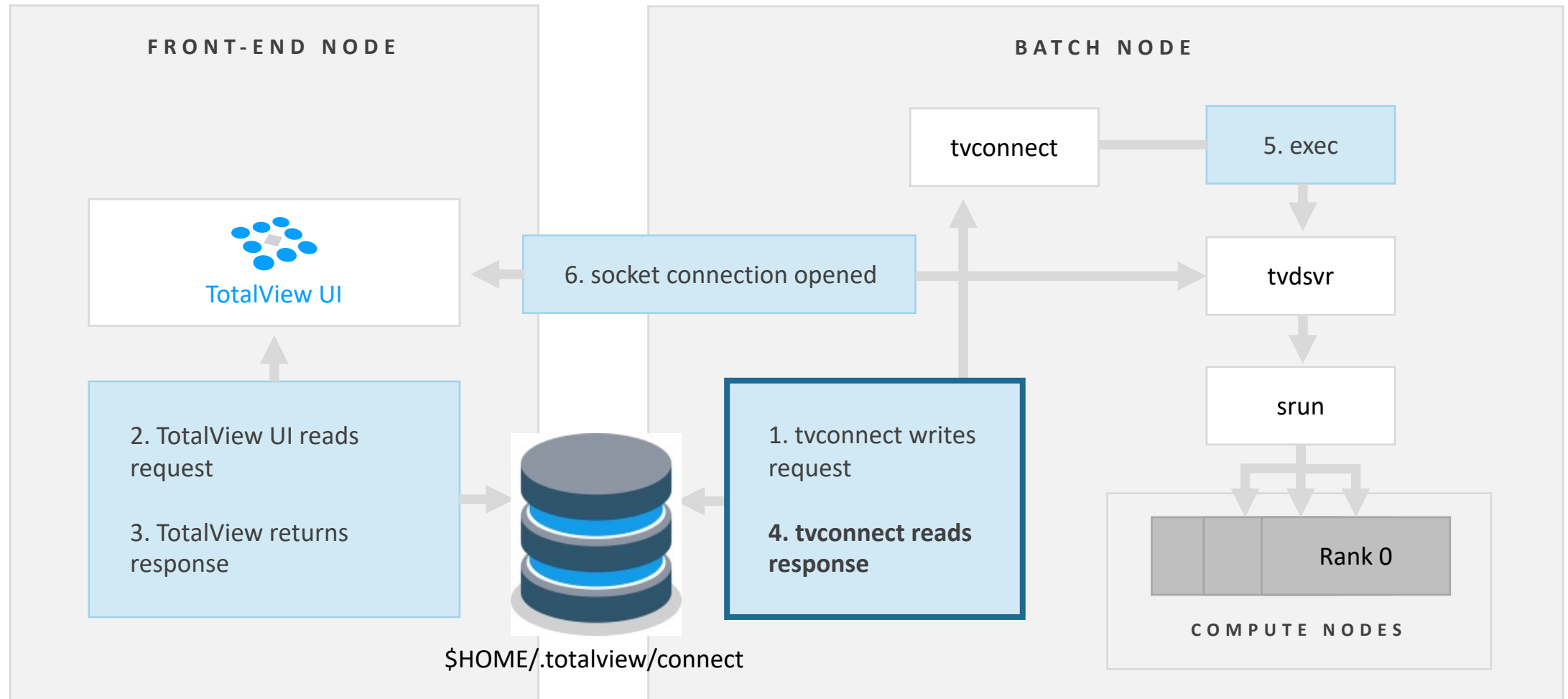
Reverse Connection Flow



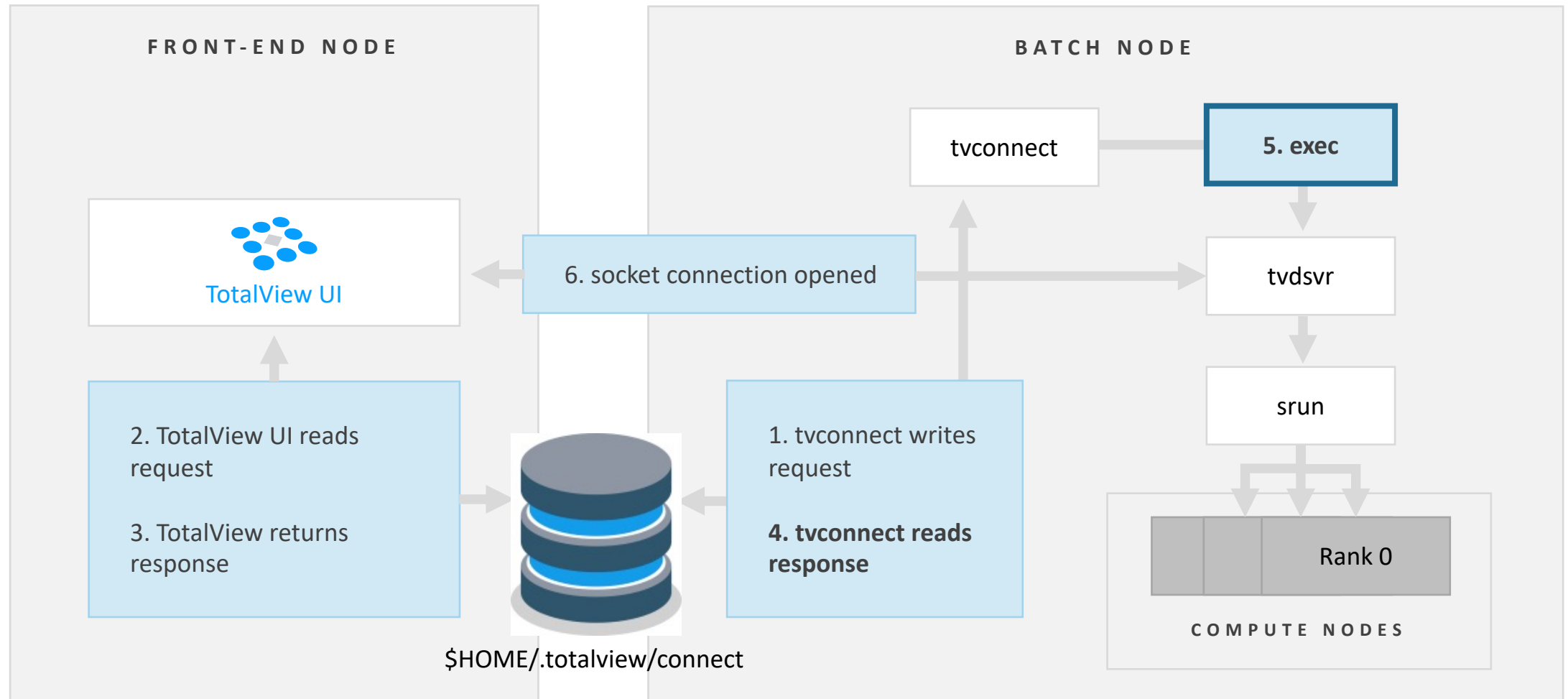
Reverse Connection Flow



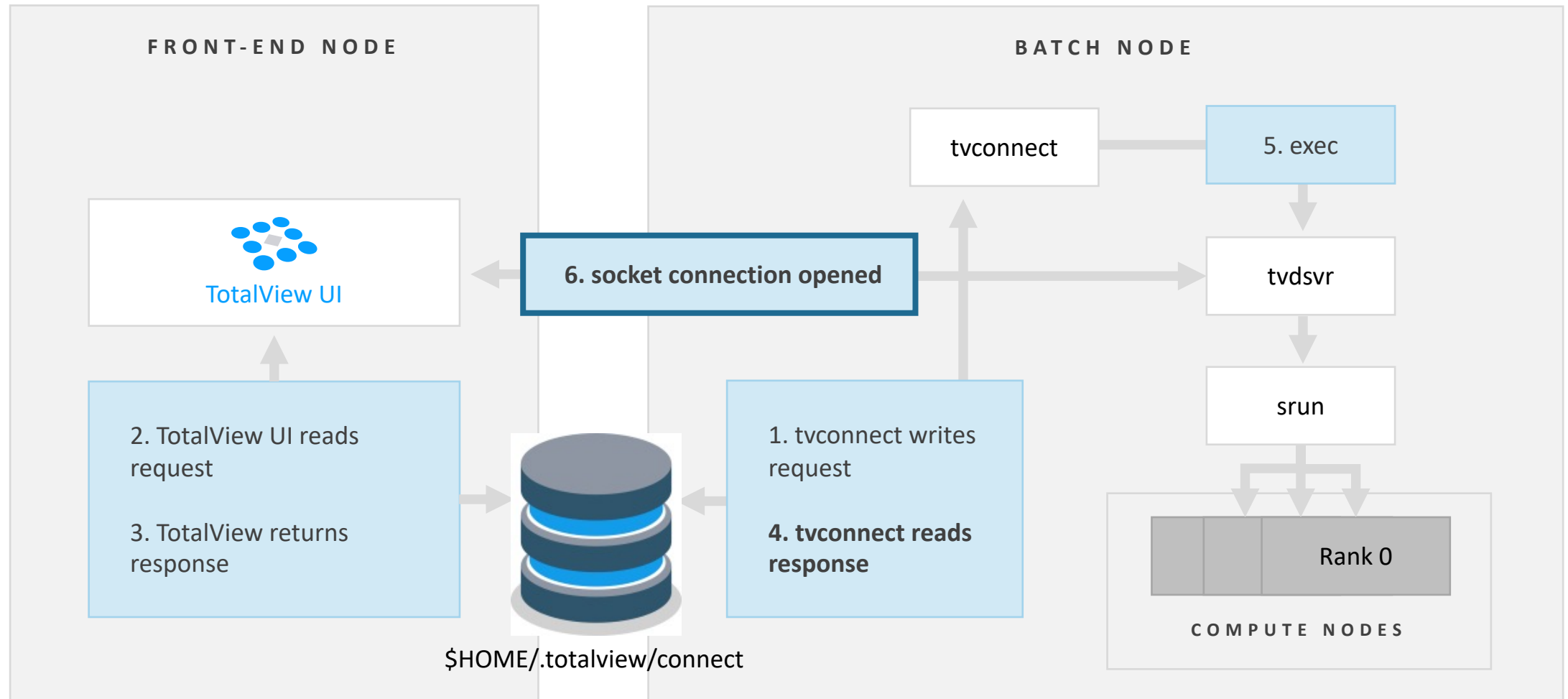
Reverse Connection Flow



Reverse Connection Flow



Reverse Connection Flow




Remote UI with Reverse Connect Demo


- TotalView Remote UI and Reverse Connect


Start Page


Start Page

What do you want to do today?


Debug a Program


Debug a Parallel Program


Attach To Process


Load Core or Replay Recording File

☒ Listen For Reverse Connections



What's New



New in TotalView 2020.2 *August 2020*



CUDA 11 Support
The TotalView 2020.2 release adds support for CUDA 11. This enables debugging support on NVIDIA's latest CUDA 11 based applications



Recent Sessions

View All

 test 
Last run on Sep 09, 2020

 demoMpi_v2 
Last run on Sep 09, 2020

 ReplayEngine_demo 
Last run on Sep 09, 2020

 ReplayEngine_demo 
Last run on Sep 09, 2020


Help/Support

Support

Find how to contact us at our Support Center.
<https://totalview.io/support>

Debugging a New Program

Session Editor

 Program Session

Session Details

Session Name

combined

Debug Options

Reverse Debugging

☐ Enable reverse debugging with ReplayEngine

Python Debugging

☐ Enable call stack filtering for Python

Standard Input Redirection

Redirect standard input from file

[Enter input file path and name]

BROWSE...

Program Details

File Name

/home/stewart/Projects/DemoDVD/programs/combined

REQUIRED

BROWSE...

Arguments

[Enter any program arguments. Ex. -option foo]

Program Environment

Environment variables for the program

[Enter line-separated NAME=VALUE pairs]

Standard Output/Error Redirection

RESET

LOAD SESSION


CANCEL

37 | TotalView by Perforce © Perforce Software, Inc.

totalview.io

Debugging a Parallel Program

Session Editor

 Parallel Session

Session Details

Session Name

Parallel Details

Parallel System

Open MPI

Tasks (-np):

16

Additional Starter Arguments

[Enter starter arguments as needed]

Standard Input Redirection

Standard Output/Error Redirection

Program Details

File Name

/home/stewart/Projects/DemoDVD/programs/demoMpi_v2

BROWSE...

Arguments

[Enter any program arguments. Ex. -option foo]

Debug Options

Reverse Debugging

☐ Enable reverse debugging with ReplayEngine

Python Debugging

☐ Enable call stack filtering for Python

Program Environment

Environment variables for the program

[Enter line-separated NAME=VALUE pairs]

RESET


LOAD SESSION

CANCEL

38 | TotalView by Perforce © Perforce Software, Inc.


totalview.io

Attach to a Running Program

 **Attach to Running Program(s)**

Session Name
[Enter or select a session name, e.g. myprogram with ReplayEngine]

Processes

Host 10.0.2.15 (local) 

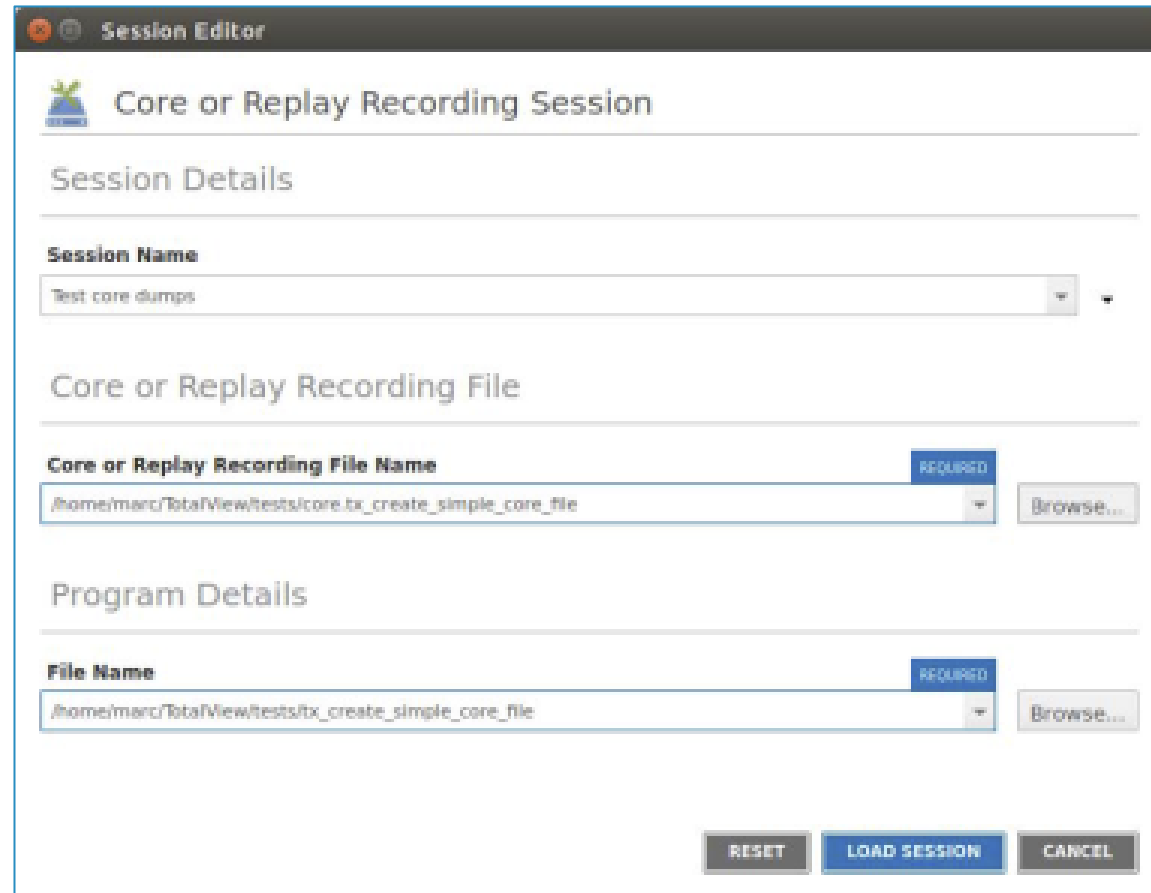
Program	State	PID	PPID	Host	Path
▼ gnome-terminal	S	1914	1208	10.0.2.15	/usr/bin/
gnome-ptty-helpe	S	1921	1914	10.0.2.15	
▼ bash	S	1922	1914	10.0.2.15	/bin/
▼ rwcresapp	S	2128	1922	10.0.2.15	/usr/toolworks/codedynamics.2015X.10.15/linux-x86-...
tvengine	R	2171	2128	10.0.2.15	/usr/toolworks/codedynamics.2015X.10.15/linux-x86-64/
▼ process_viewer	R	1383	1308	10.0.2.15	/usr/bin/

PID [Enter PID, if not in the list] **Program** [Enter program path and name, e.g. /home/smith/mypro...

Debug Options ▲

Reverse Debugging
☐ Enable reverse debugging with ReplayEngine

Open a Core File or Replay Recording Session



The screenshot shows a window titled "Session Editor" with a dark header bar. Below the header, there is a section titled "Core or Replay Recording Session" with a small icon. The main content area is divided into three sections: "Session Details", "Core or Replay Recording File", and "Program Details".

Session Details

Session Name

Test core dumps

Core or Replay Recording File

Core or Replay Recording File Name REQUIRED

/home/marc/totalView/tests/core.tx_create_simple_core_file

Browse...

Program Details

File Name REQUIRED









/home/marc/totalView/tests/tx_create_simple_core_file

Browse...

RESET LOAD SESSION CANCEL

Starting a Previous Debugging Session

Recent Sessions

	Test Core Simple 
Last run on May 04, 2015	
	tsl_core1 
Last run on May 04, 2015	
	Test Core 
Last run on May 04, 2015	
	Attach Test 2 
Last run on May 04, 2015	

UI Navigation and Process Control

TotalView's Default Views

Processes &
Threads View

Lookup File or
Function

Documents

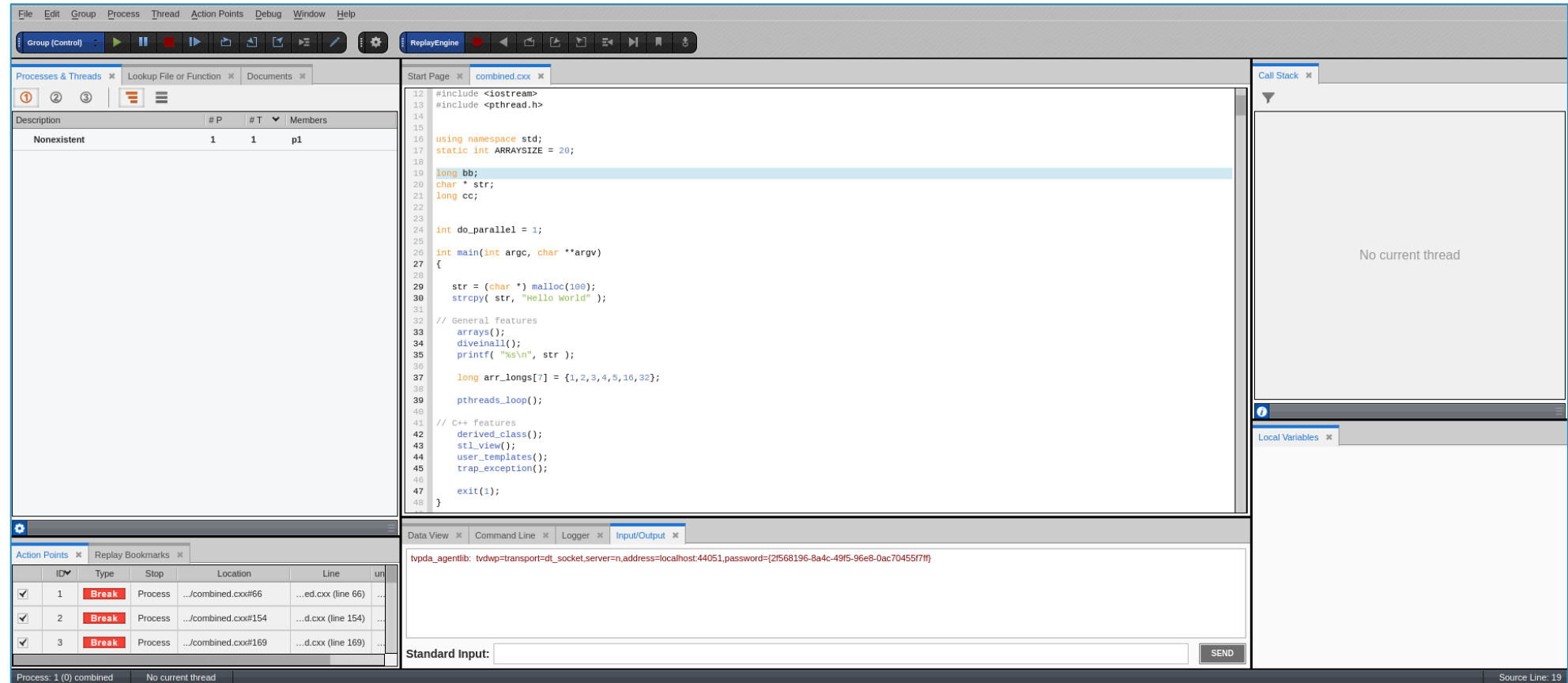
Source View

Call Stack View

Local Variables
View

Data View,
Command Line,
Logger,
Input/Output

Action Points,
Replay Bookmarks



Process and Threads View

Processes & Threads x

①

②

③

Description	# P	# T	Members
tx_lok_loop (53)	4	4	p1-4
Mixed	1	1	p1
<unknown address>	1	1	p1.1
1.1	1	1	p1.1
Running	1	1	p1.1
_clone	1	2	p1.2-3
1.2	1	1	p1.2
Stopped	1	1	p1.2
1.3	1	1	p1.3
Stopped	1	1	p1.3
Breakpoint	3	3	p2-4
store	3	3	p3.1, p2.2, p4.2
2.2	1	1	p2.2
Breakpoint	1	1	p2.2
3.1	1	1	p3.1
Breakpoint	1	1	p3.1
4.2	1	1	p4.2
Breakpoint	1	1	p4.2
_clone	3	6	p2.1, p4.1, p3.2, p2.4.3

Select process or thread attributes to group by:

☐ Control Group

☒ Share Group

☒ Process State

☒ Function

☒ Thread ID

☒ Thread State

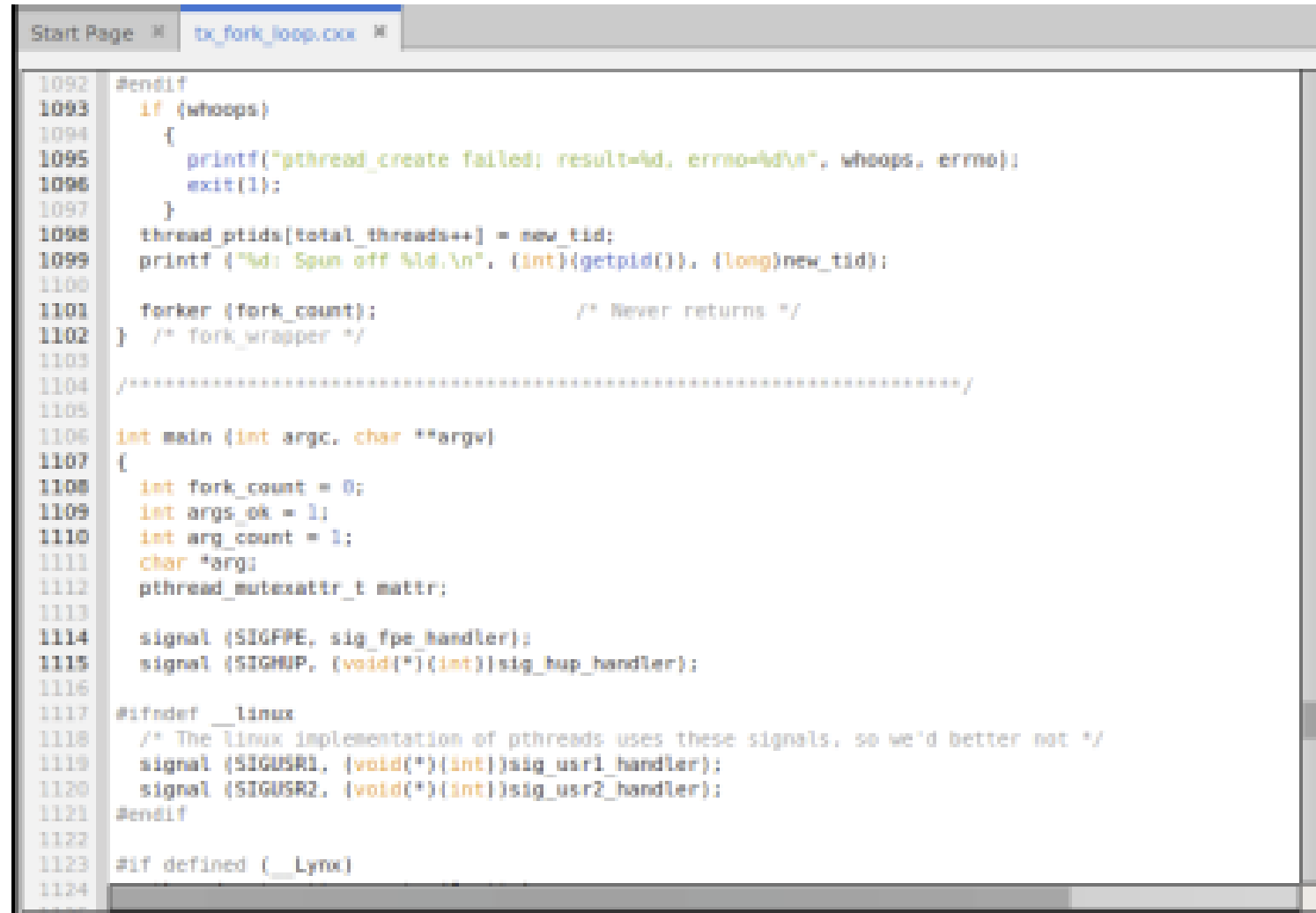
☐ PC

↑

↺

↓

Source View



```
1092 #endif
1093     if (whoops)
1094     {
1095         printf("pthread_create failed: result=%d, errno=%d\n", whoops, errno);
1096         exit(1);
1097     }
1098     thread_ptids[total_threads++] = new_tid;
1099     printf ("%d: Spun off %d.\n", (int){getpid()}, (long)new_tid);
1100
1101     forker (fork_count);          /* Never returns */
1102 } /* fork_wrapper */
1103
1104 /*****
1105
1106 int main (int argc, char **argv)
1107 {
1108     int fork_count = 0;
1109     int args_ok = 1;
1110     int arg_count = 1;
1111     char *arg;
1112     pthread_mutexattr_t mattr;
1113
1114     signal (SIGFPE, sig_fpe_handler);
1115     signal (SIGHUP, (void (*)(int))sig_hup_handler);
1116
1117     #ifndef __linux
1118     /* The linux implementation of pthreads uses these signals, so we'd better not */
1119     signal (SIGUSR1, (void (*)(int))sig_usr1_handler);
1120     signal (SIGUSR2, (void (*)(int))sig_usr2_handler);
1121     #endif
1122
1123     #if defined (__Lynx)
1124
```


Call Stack View and Local Variables Panel

Call Stack

C++

funcA

C++

main

__libc_start_main

_start

Function

funcA

Source

...ewart/Projects/DemoDVD/src/ReplayEngine_demo.cxx

Line

35

FP

0x7ffe94b13d00

Local Variables

Name	Type	Value
Arguments		
a	int	0x00000000 (0)
Block at Line 35		
b	int	0x00000000 (0)

Action Points and Replay Engine Bookmarks

Action Points					
Replay Bookmarks					
	ID	Type	Stop	Location	Line
<input checked="" type="checkbox"/>	1	Break	Process	..._demo.cxx#25	...o.cxx (line 25)
<input checked="" type="checkbox"/>	2	Watch	Group	...058 (arraylength)	

Replay Bookmarks				
ID	PC	File	Line	Comment
1	0x 4007bb	badstuff	61	error_loc

Data View, Command Line and Logger

The screenshot displays the TotalView interface with three overlapping panels. The top panel is the 'Data View' tab, showing a table with columns 'Name', 'Type', 'Thread ID', and 'Value'. It contains one entry: 'a' of type 'int' on thread '1.1' with value '0x00000000 (0)'. Below the table is a button '[Add New Expression]'. The middle panel is the 'Command Line' tab, showing system messages: 'Linux x86_64 TotalView 2019.3.14', 'Thread 1.1 has appeared', 'Created process 1 (4343), named "ReplayEngine_demo"', 'Thread 1.1 hit breakpoint 1 at line 25 in "main"', 'Thread 1.1 received a signal (Segmentation violation)', and 'Process 1 has exited'. The bottom panel is the 'Logger' tab, showing detailed logs: 'Indexing 23540 bytes of DWARF '.eh_frame' symbols from '/lib64/libm.so.6'...', 'done', 'Library /lib64/libgcc_s.so.1, with 2 asects, was linked at 0x00000000, and initially loaded at 0xff000000903f7700', 'Mapping 2214 bytes of ELF string data from '/lib64/libgcc_s.so.1'...', 'done', 'Indexing 6732 bytes of DWARF '.eh_frame' symbols from '/lib64/libgcc_s.so.1'...', 'done', 'WARNING: (instance 1) Setting a breakpoint at an absolute mode address! This is not usually correct.', and '::temp_bp_t: enable tmp: Operation Failed'.

Name	Type	Thread ID	Value
a	int	1.1	0x00000000 (0)

[Add New Expression]

Linux x86_64 TotalView 2019.3.14

Thread 1.1 has appeared
Created process 1 (4343), named "ReplayEngine_demo"
Thread 1.1 hit breakpoint 1 at line 25 in "main"
Thread 1.1 received a signal (Segmentation violation)
Process 1 has exited

Created process 1 (4344)
Thread 1.1 has appeared
Thread 1.1 hit breakpoint 1 at line 25 in "main"
d1.<>

Indexing 23540 bytes of DWARF '.eh_frame' symbols from '/lib64/libm.so.6'...
done

Library /lib64/libgcc_s.so.1, with 2 asects, was linked at 0x00000000
, and initially loaded at 0xff000000903f7700

Mapping 2214 bytes of ELF string data from '/lib64/libgcc_s.so.1'...
done

Indexing 6732 bytes of DWARF '.eh_frame' symbols from '/lib64/libgcc_s.so.1'...
done

WARNING: (instance 1) Setting a breakpoint at an absolute mode address! This is not usually correct.

::temp_bp_t: enable tmp: Operation Failed

Lookup File or Function

The screenshot displays the TotalView IDE interface. The main editor window on the left shows a C source file with the following code:

```
097 /* Read a term which is a factor and optional * or / of a second factor. */
098
099 node_t *term ()
100 {
101     node_t *node = factor ();
102     while (nextchar == '*' || nextchar == '/') {
103         node_class_t node_class = (node_class_t) readchar ();
104         node = new_node (node_class, node, factor ());
105     }
106     return (node);
107 } /* term */
108
109 /*****
110 /* Read an expression which is a term and optional + or - of a second term. */
111
112 node_t *expression ()
113 {
114     node_t *node = term ();
115     while (nextchar == '+' || nextchar == '-') {
116         node_class_t node_class = (node_class_t) readchar ();
117         node = new_node (node_class, node, term ());
118     }
119     return (node);
120 } /* expression */
121
122 /*****
123 /* Read an expression and return an expression tree. */
124
125 node_t *readexpr ()
126 {
127     node_t *node = 0;
128     readchar ();
129     if (nextchar != EOF) {
130         while (nextchar == '\n')
131             readchar ();
132         node = expression ();
133         if (nextchar == '\n' && nextchar != ',' && nextchar != EOF)
```

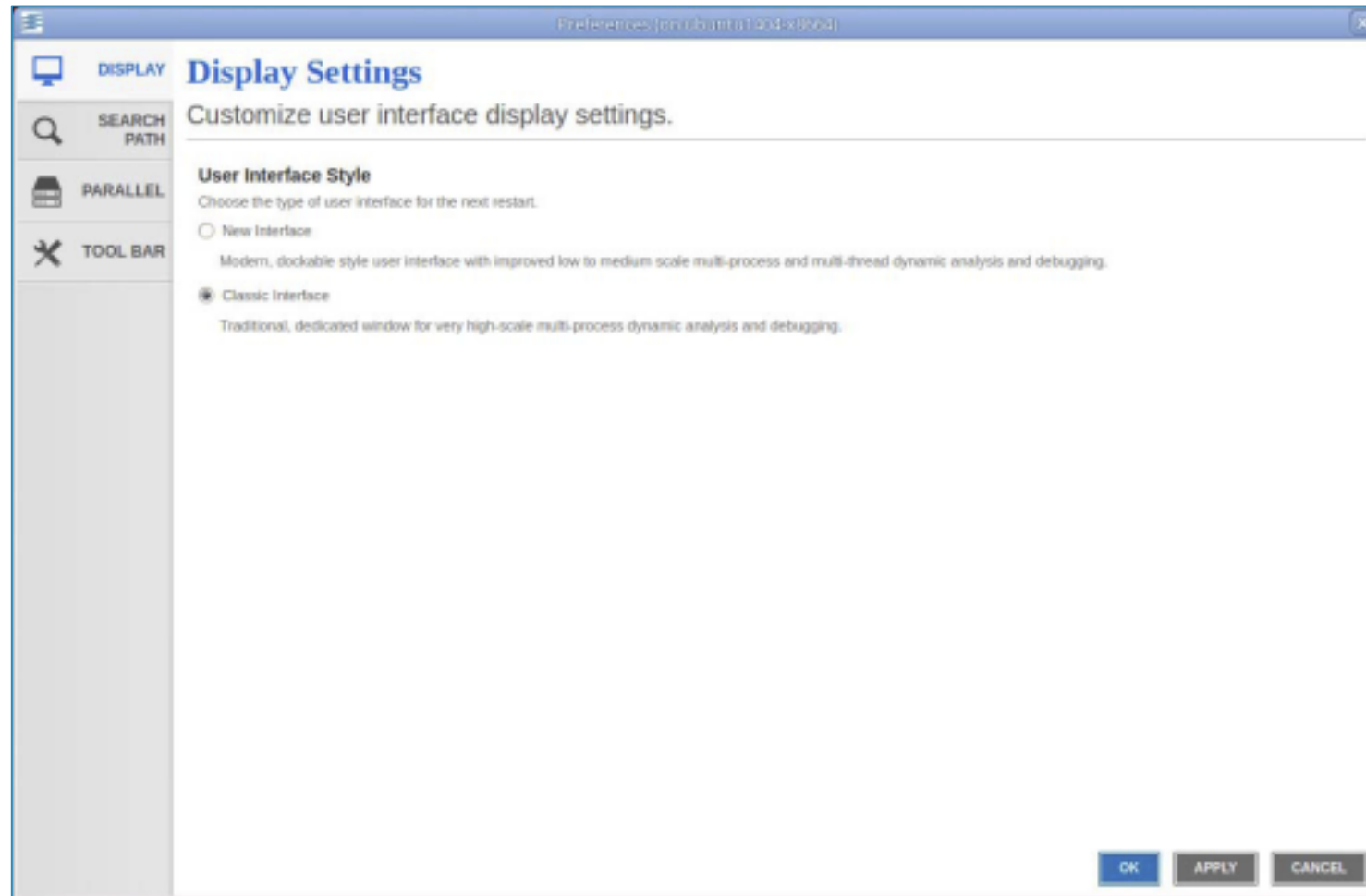
The right-hand pane is titled "Lookup File or Function". It contains a search input field with the text "ex". Below the input, it states "Matching Items: Found 7 matching results". A list of items is shown:

- expr
- readexpr
- expression
- readexpr.c
- evalexpr.c
- error
- evaluate

The item "expression" is highlighted with a blue background. At the bottom of the pane, there is a checkbox labeled "Display full path information" which is currently unchecked.

Preferences

Display Settings

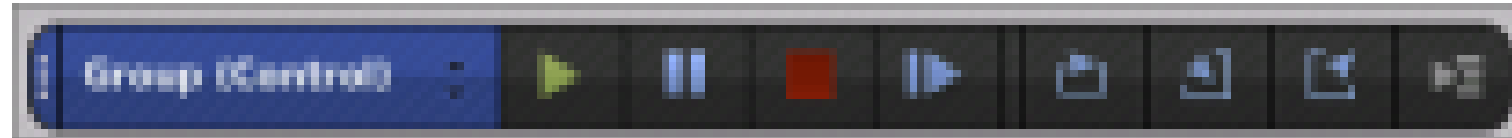



TotalView Toolbar



Command	Description
Go	Sets the thread to running until it reaches a stopping point. Often this will be a breakpoint that you have set, but the thread could stop for other reasons.
Halt	Stops the thread at its current execution point.
Kill	Stops program execution. Existing breakpoints and other settings remain in effect.
Restart	Stops program execution and restarts the program from the beginning. Existing breakpoints and other settings remain in effect. This is the same as clicking Kill followed by Go.
Next	Moves the thread to the next line of execution. If the line the thread was on includes one or more function calls, TotalView does not step into these functions but just executes them and returns.
Step	Like Next, except that TotalView does step into any function calls, so the thread stops at the first line of execution of the first function call.
Out	If the thread is in a block of execution, runs the thread to the first line of execution beyond that block.
Run To	If there is a code line selected in one of the Source views, the thread will stop at this line, assuming of course that it ever makes it there. This operates like a one-time, temporary breakpoint.

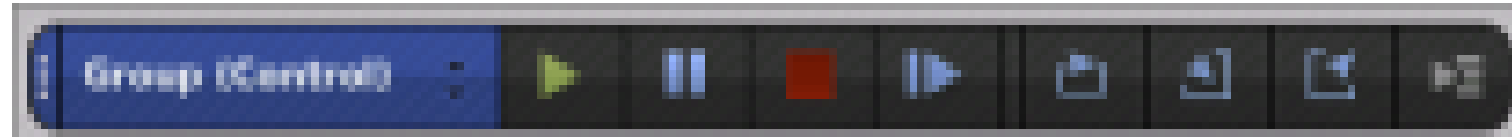
Stepping Commands





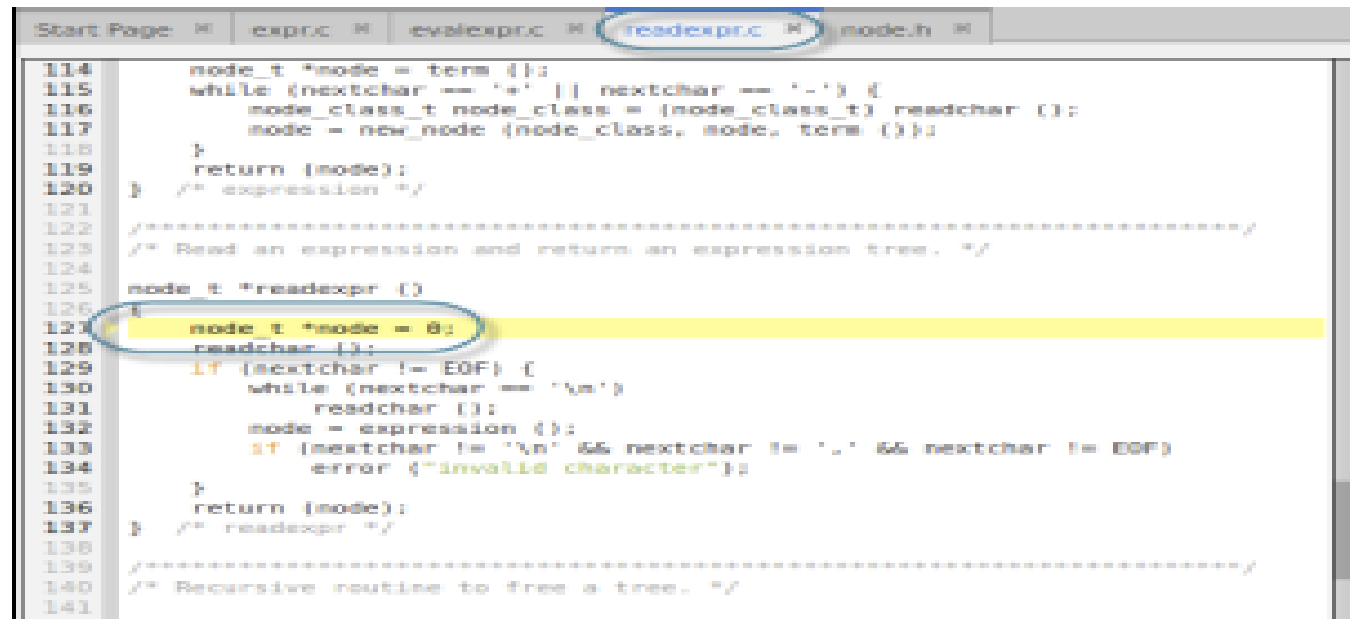
Select Step  in the toolbar. TotalView stops the program just before the first executable statement, the call to `setjmp (context);`

```
27 int main (int argc, char **argv)
28 {
29     node_t *node;
30     setjmp (context);
31     while (node = readexpr ()) {
32         previous = evaluate (node);
33         printf ("%g %ld (%x%lx)\n", previous, (long) previous, (long)
34             fflush (stdout);
35         freetree (node);
36     }
```

Stepping Commands



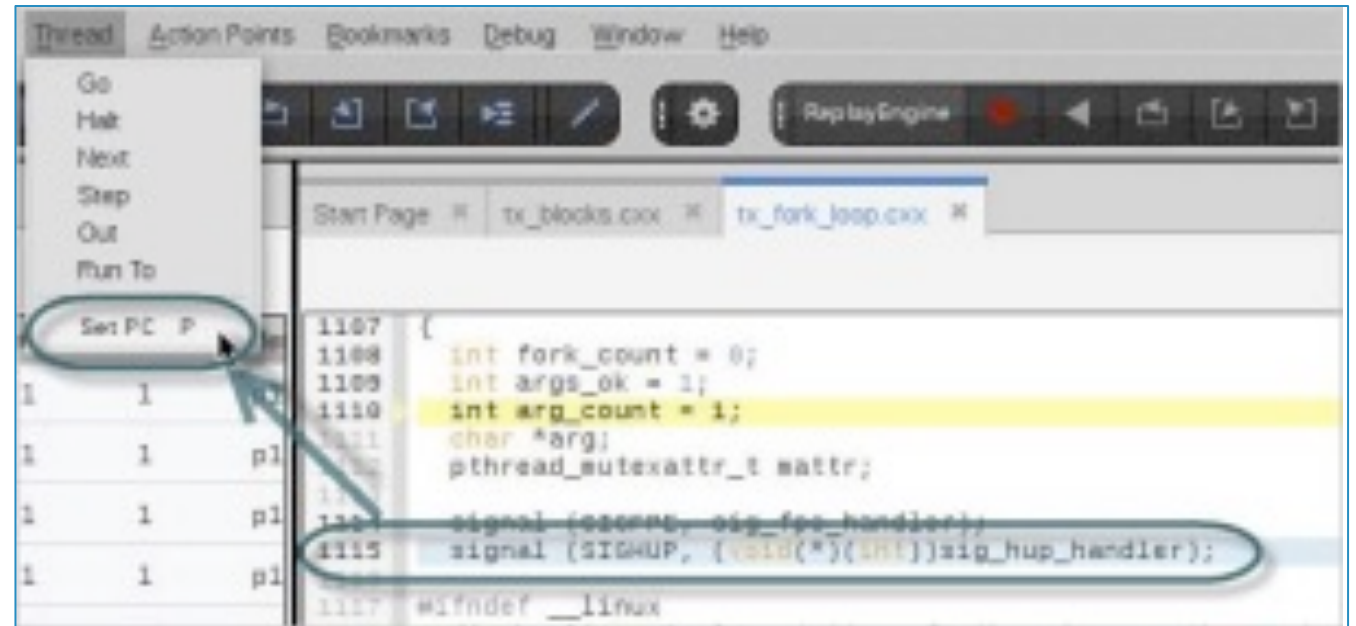
Select Step  again to advance to the while loop on line 31, and then select Step  again to *step into* the readexpr() function. (Next would *step over*, or execute it).



```
114     node_t *node = term ();
115     while (nextchar == 'e' || nextchar == '-') {
116         node_class_t node_class = (node_class_t) readchar ();
117         node = new_node (node_class, node, term ());
118     }
119     return (node);
120 } /* expression */
121
122 /*****
123  /* Read an expression and return an expression tree. */
124
125 node_t *readexpr ()
126 {
127     node_t *node = 0;
128     readchar ();
129     if (nextchar != EOF) {
130         while (nextchar == '\n')
131             readchar ();
132         node = expression ();
133         if (nextchar != '\n' && nextchar != '.' && nextchar != EOF)
134             error ("invalid character");
135     }
136     return (node);
137 } /* readexpr */
138
139 /*****
140  /* Recursive routine to free a tree. */
141
```


Using Set PC

- Resumes execution from an arbitrary point
- Select the line
- Thread->Set PC



Demo

- TotalView UI demo (QT Threads Example)

Action Points

<div><div></div><div></div></div>					
<div><div>Action Points ✕</div><div>Replay Bookmarks ✕</div></div>					
	ID▼	Type	Stop	Location	Line
<input checked="" type="checkbox"/>	1	Break	Process	..._demo.cxx#25	...o.cxx (line 25
<input checked="" type="checkbox"/>	2	Watch	Group	...058 (arraylength)	

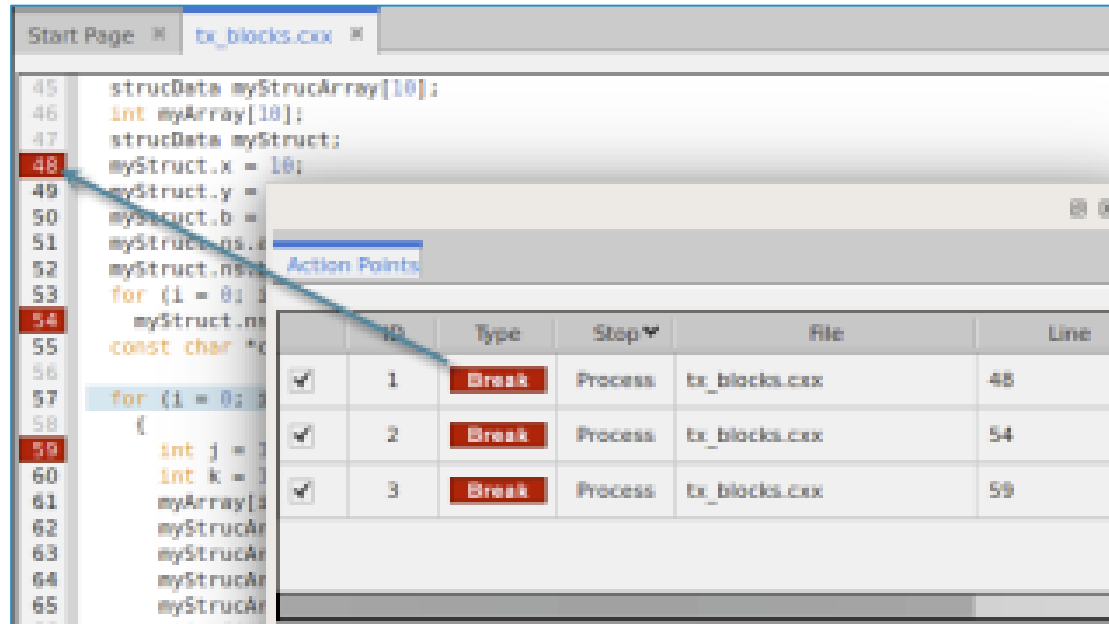
Breakpoint

Evalpoint

Watchpoint

Barrierpoint

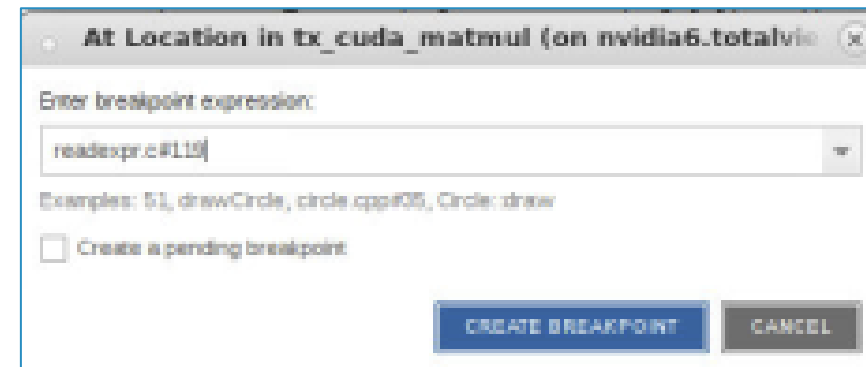
Setting Breakpoints



- Setting action points
 - Single-click line number
 - Right clicking on the line number and using the context menu
 - Clicking a line in the source view then selecting the Action Points -> Set breakpoint menu option

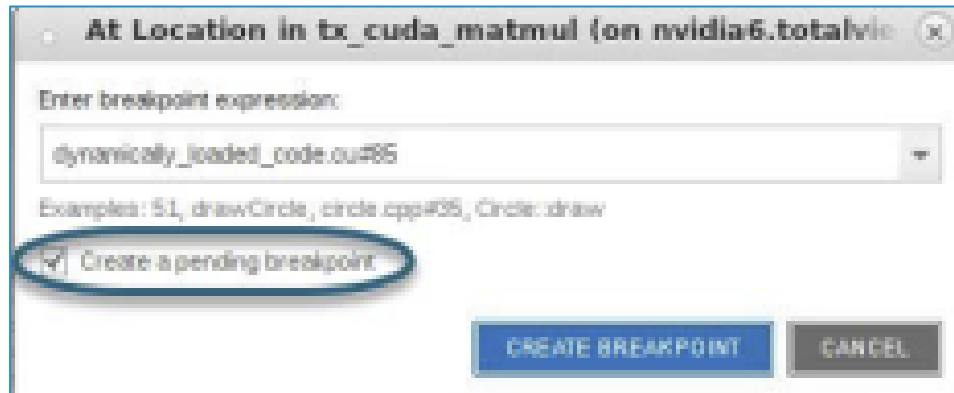
Setting Breakpoints

- Breakpoint->At Location...
 - Specify function name or line number
 - If function name, TotalView sets a breakpoint at first executable line in the function

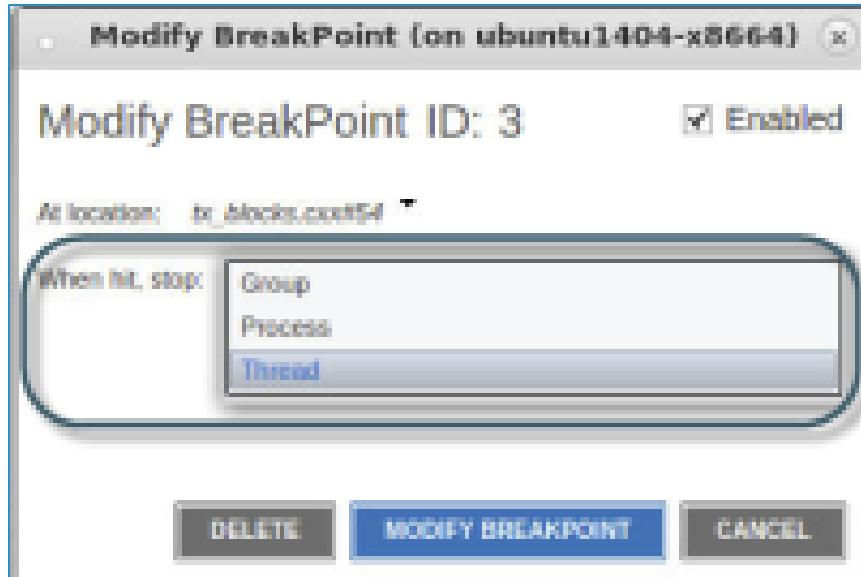


Pending Breakpoints

- Useful when setting a breakpoint on a library that has not yet been loaded into memory



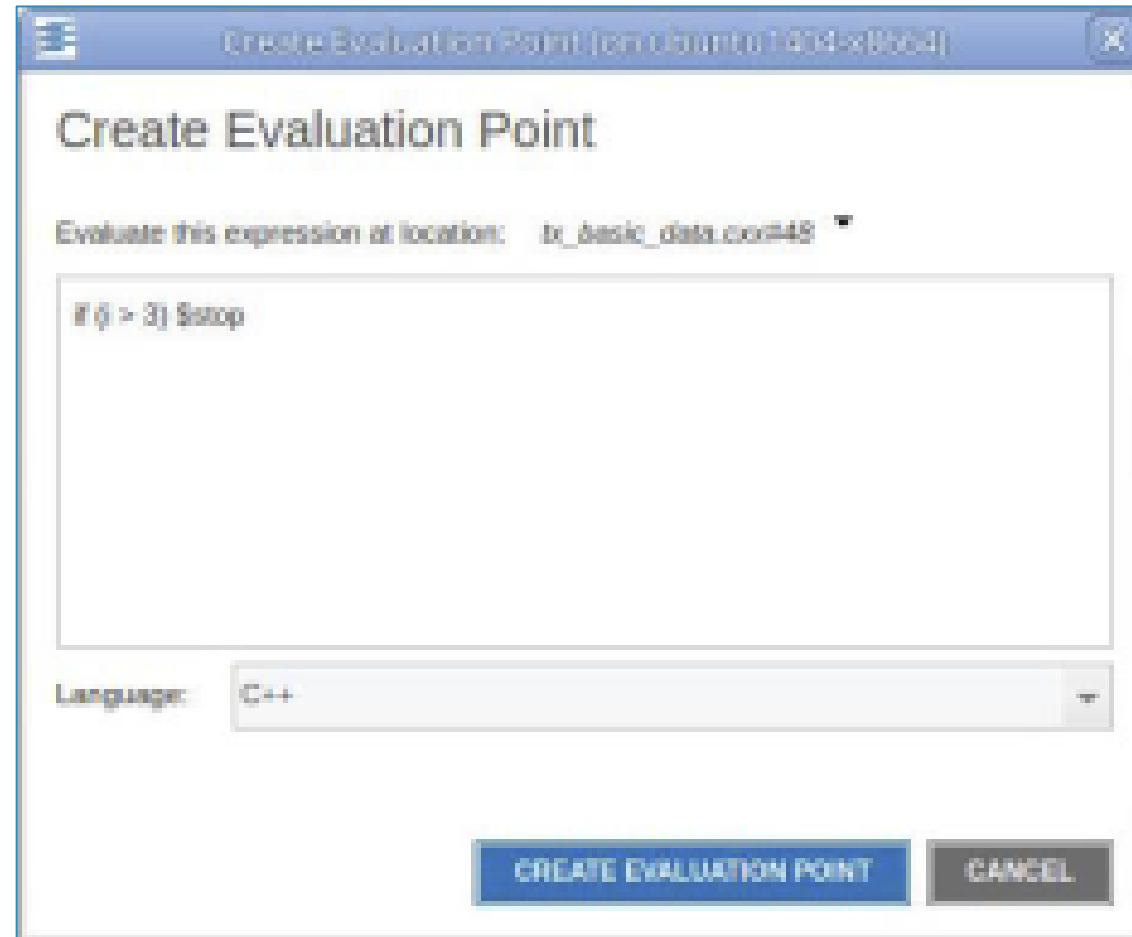
Modifying Breakpoints



- Enable / Disable / Delete a breakpoint
- Adjust the breakpoints width

- **Group:** Stops all running threads in all processes in the group.
- **Process:** Stops all the running threads in the process containing the thread that hit the breakpoint
- **Thread:** Stops only the thread that first executes to this breakpoint

Evalpoints



Evalpoints

- Use Eval points to :
 - Include instructions that stop a process and its relatives
 - Test potential fixes or patches for your program
 - Include a goto for C or Fortran that transfers control to a line number in your program
 - Execute a TotalView function
 - Set the values of your program's variables

Evalpoints Examples

- Print the value of a variable to the command line

```
printf("The value of result is %d\n", result);
```

- Skip some code

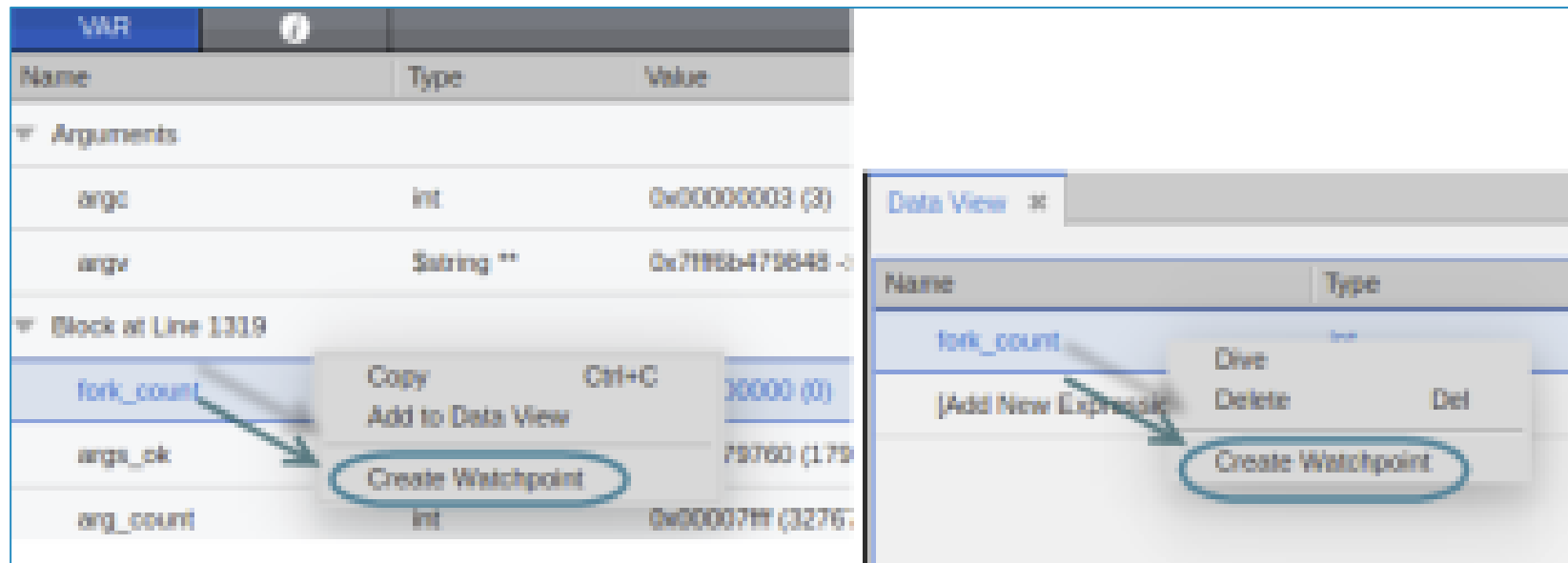
```
goto 63;
```

- Stop a loop after a certain number of iterations

```
if ( (i % 100) == 0)
{
    printf("The value of i is %d\n", i);
    $stop;
};
```

Watchpoints

- Watchpoints are set on a specific memory location
- Execution is stopped when the value stored in that memory location changes
- A breakpoint stops **before** an instruction executes. A watchpoint stops **after** an instruction executes



Using Watchpoint Expressions

- TotalView has two variables that are used exclusively with watchpoint expressions:
 - \$oldval: The value of the memory locations before a change is made.
 - \$newval: The value of the memory locations after a change is made.

- Example 1

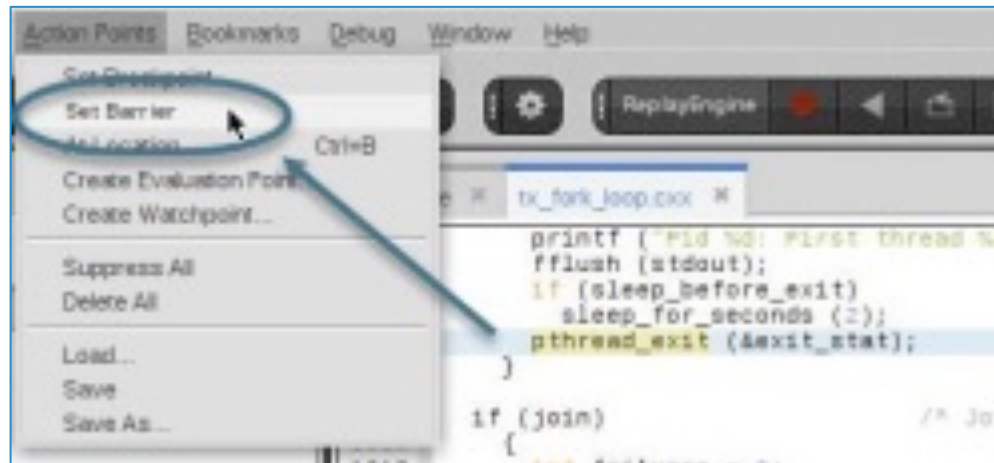
```
if (iValue != 42 && iValue != 44) {  
    iNewValue = $newval; iOldValue = $oldval; $stop;}
```

- Example 2

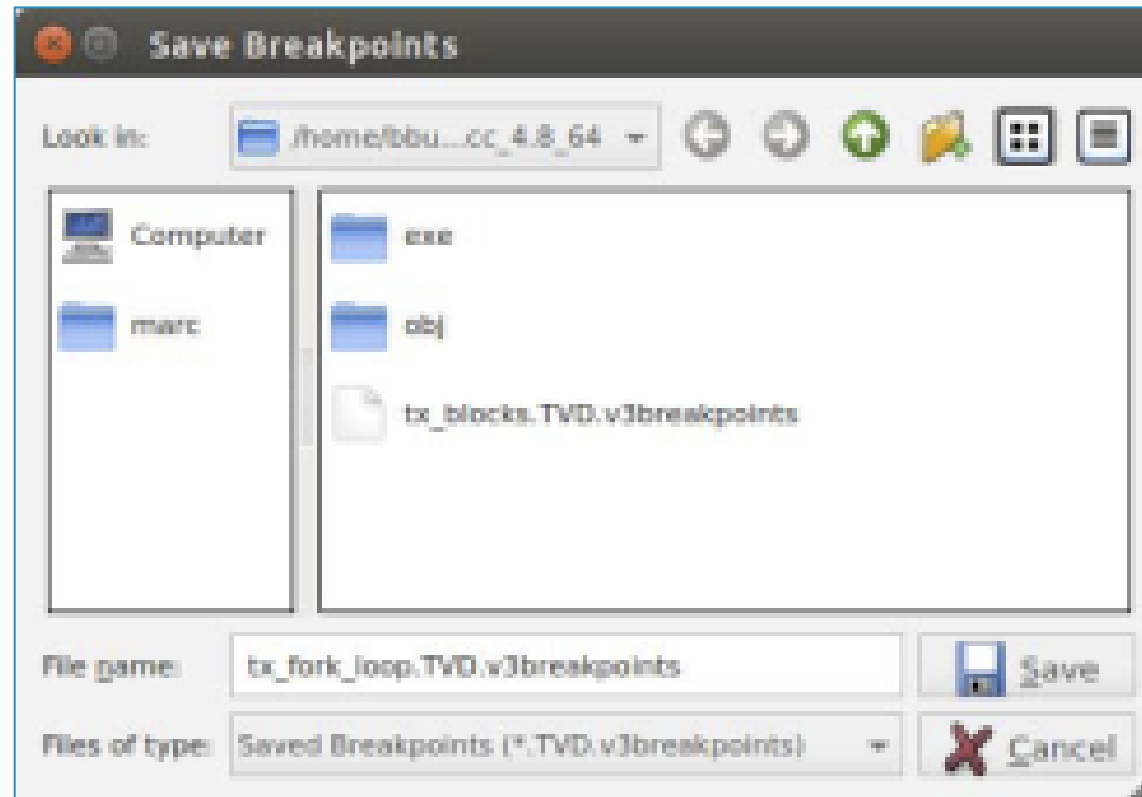
```
if ($oldval >= 0 && $newval < 0) $stop
```

Barrier Breakpoints

- Used to synchronize a group of threads or processes defined in the action point
- Threads or processes are held at barrierpoint until all threads or processes in the group arrive
- When all threads or processes arrive the barrier is satisfied and the threads or processes are released



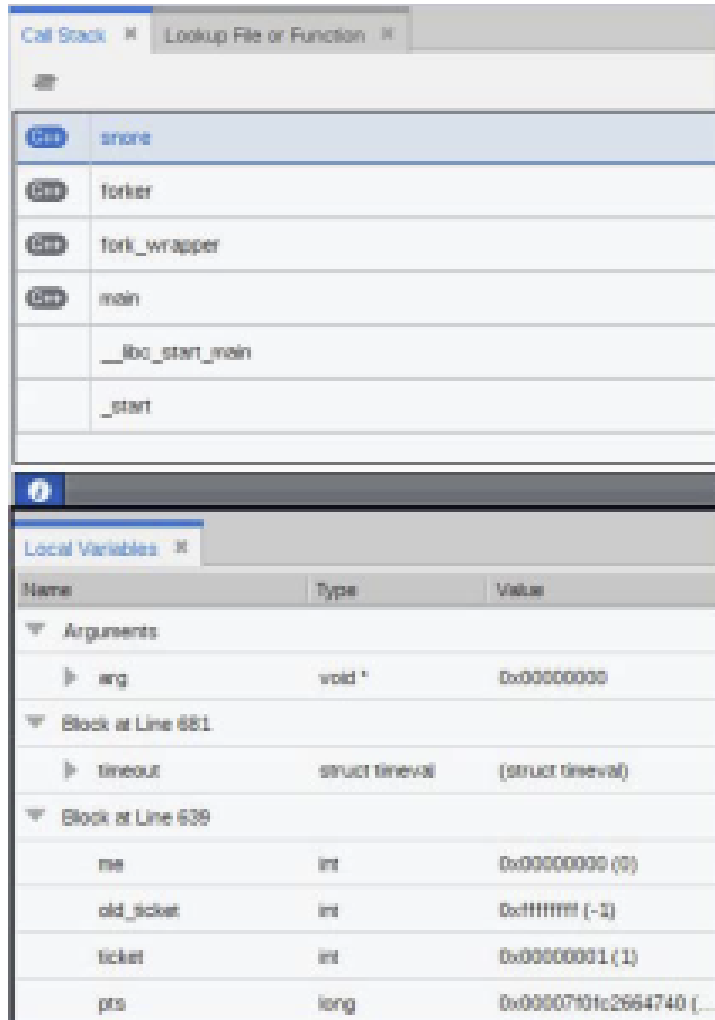
Saving Breakpoints



From the Action Points menu select Save or Save As to save breakpoints
Turn on option to save action points on exit

Examining and Editing Data

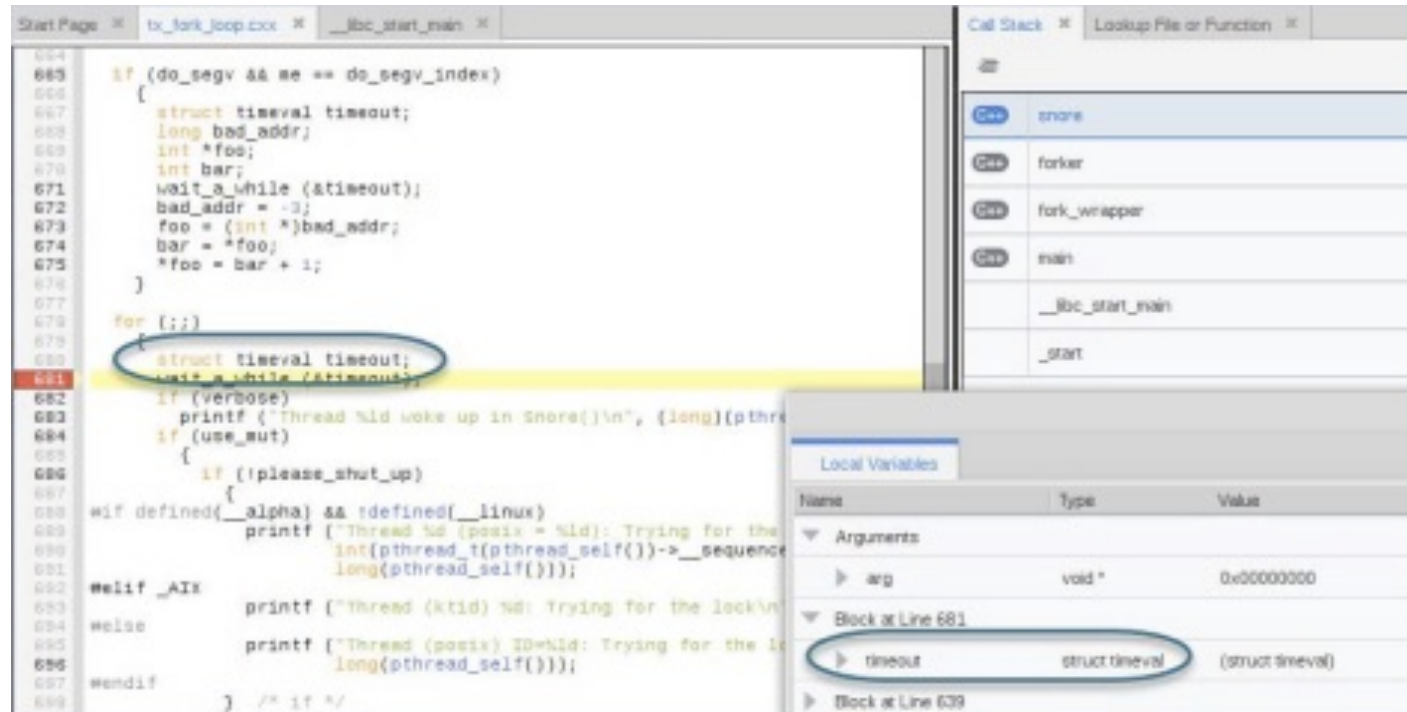
Call Stack and Local Variables



The Call Stack View consists of two panels :

- The Call Stack panel
- The Local Variables panel

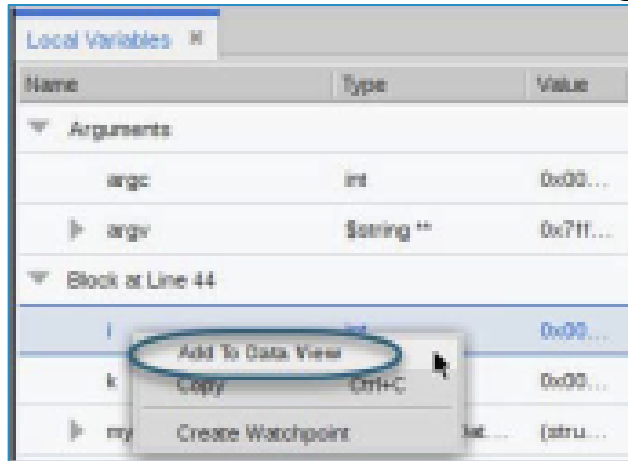
The Local Variables Panel



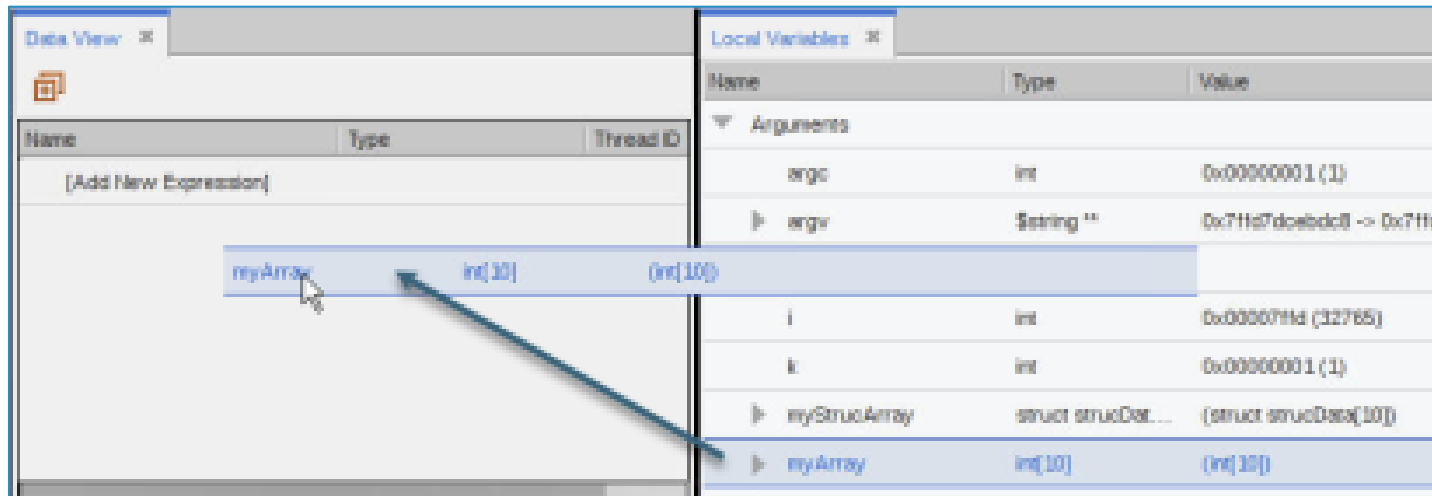
The Local Variables panel displays variables for the thread of interest (TOI)

The Data View Panel

Add data to the Data View using the context menu or by dragging and dropping



Context menu

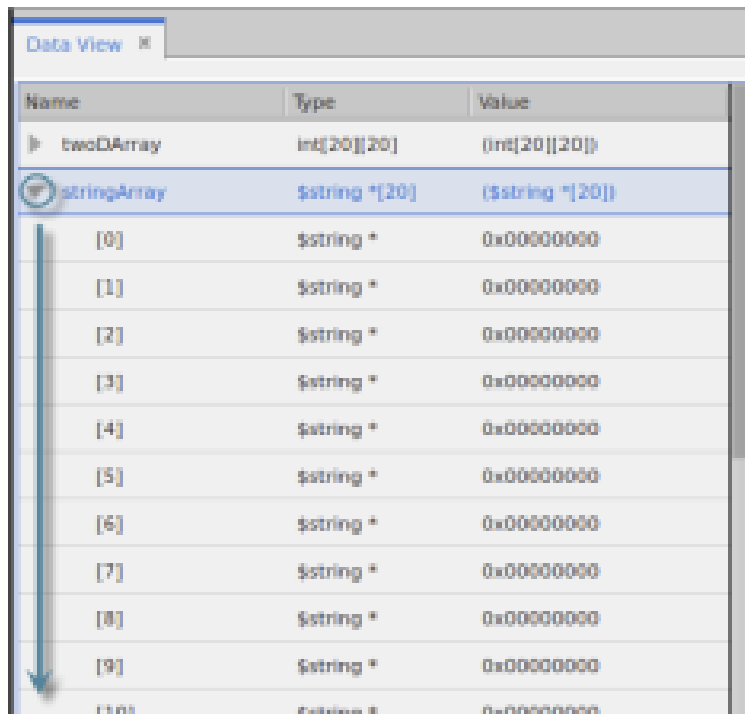


Drag and drop

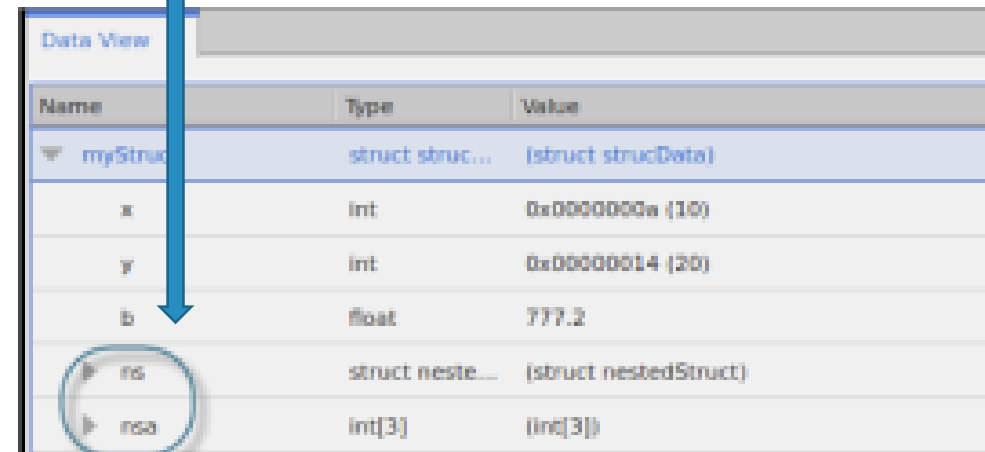
The Data View Panel

Select the right arrow to display the substructures in a complex variable

Any nested structures are displayed in the data view



Name	Type	Value
twoDArray	int[20][20]	(int[20][20])
stringArray	\$string *[20]	(\$string *[20])
[0]	\$string *	0x00000000
[1]	\$string *	0x00000000
[2]	\$string *	0x00000000
[3]	\$string *	0x00000000
[4]	\$string *	0x00000000
[5]	\$string *	0x00000000
[6]	\$string *	0x00000000
[7]	\$string *	0x00000000
[8]	\$string *	0x00000000
[9]	\$string *	0x00000000
[10]	\$string *	0x00000000




Name	Type	Value
myStruct	struct struc...	(struct strucData)
x	int	0x0000000a (10)
y	int	0x00000014 (20)
b	float	777.2
ns	struct neste...	(struct nestedStruct)
nsa	int[3]	(int[3])

The Data View Panel

Dive on a single element to view individual data in the Data View

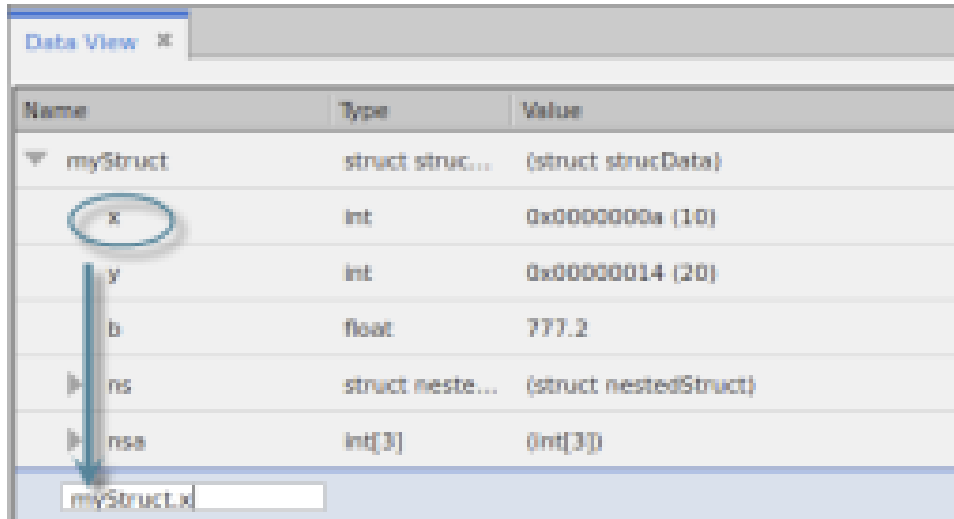
[5]	\$string *	0x00000000
[6]	\$string *	0x00000000
[7]	\$string *	
[8]	\$string *	



[18]	\$string *	0x00000000
[19]	\$string *	0x00000000
▶ <u>stringArray[7]</u>	\$string *	0x00000000
[Add New Expression]		

The Data View Panel

Enter a new expression in the Data View panel to view that data



Name	Type	Value
myStruct	struct struc...	(struct structData)
x	int	0x0000000a (10)
y	int	0x00000014 (20)
b	float	777.2
ns	struct neste...	(struct nestedStruct)
nsa	int[3]	(int[3])

myStruct.x

Type the expression in the [Add New expression] field

myStruct.x	int	0x0000000a (10)
------------	-----	-----------------

A new expression is added

myStruct.x + 5	int	0x0000000f (15)
----------------	-----	-----------------

Increment a variable

The Data View Panel

Dereferencing a pointer

Name	Type	Value
localString	\$string *	0x004009c8 -> "HelloHelloHelloHelloHel..."

When you dive on a variable, it is not dereferenced automatically

Name	Type
localString	\$string

Double click in the Name column to make it editable and dereference the pointer

Name	Type	Value
*localString	\$string	"HelloHelloHelloHelloHelloHello"

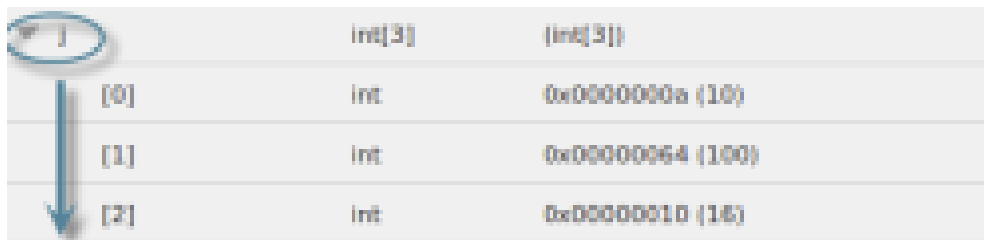
The Data View displays the variables value

The Data View Panel

Casting to another type



Cast a variable into an array by adding the array specifier

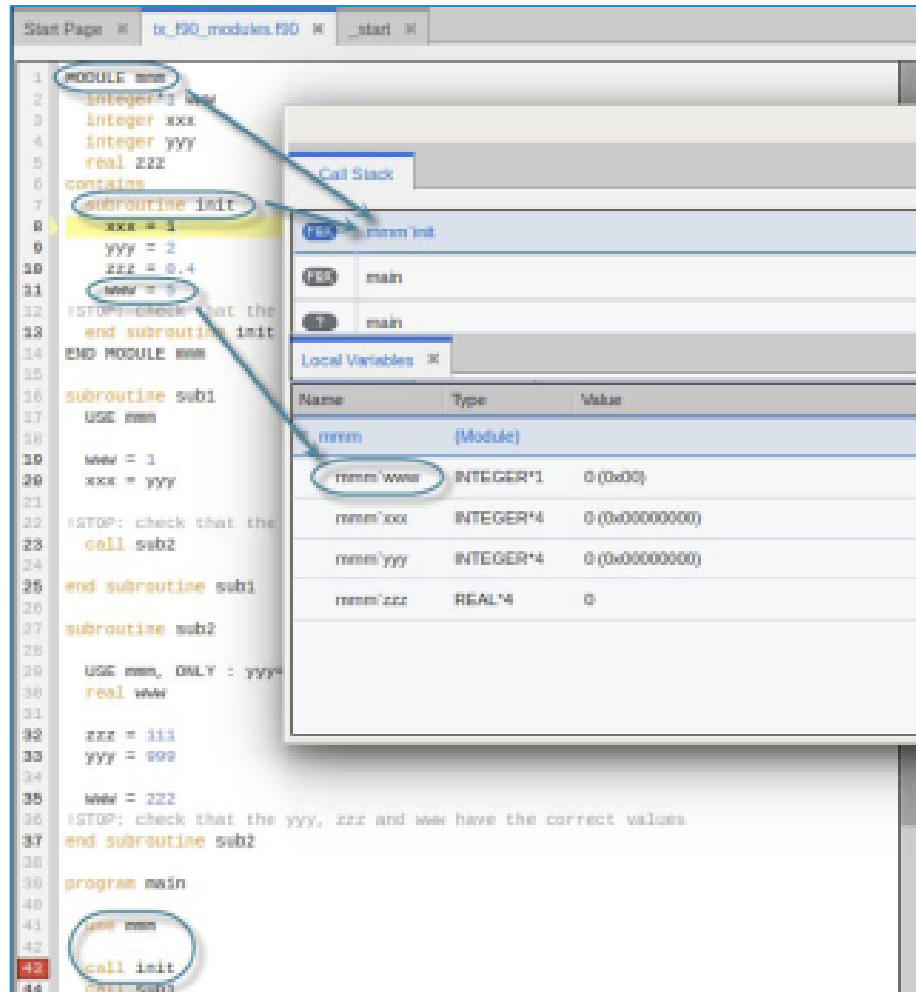


A screenshot of the Data View Panel showing the array view of variable 'j'. The variable 'j' is circled, and a blue arrow points down to the array view. The array view shows three elements, each of type 'int'.

	int[3]	(int[3])
[0]	int	0x0000000a (10)
[1]	int	0x00000064 (100)
[2]	int	0x00000010 (16)

TotalView displays the array

Viewing Data in Fortran



The qualified subroutine name appears in the Call Stack view.

The qualified variable names appear in the Local Variable panel.

Fortran Common Blocks

The screenshot displays the TotalView IDE interface. The main editor shows a Fortran program named `common_test`. The code includes variable declarations for `ix`, `iy`, `iz`, `fo`, `foo`, `ix_x`, `iy_y`, and `iz_z`. It also features a `common` block with members `fo_o`, `ix_x`, `iy_y`, and `iz_z`. A `real` array `xarray` is declared and allocated. The `Call Stack` panel on the right shows the current function `common_test`. The `Local Variables` panel on the right lists variables: `fo` (INTEGER*4), `xarray` (REAL*8, allocatable), `fo_o` (Common), `fo_o` (Common), `ix` (REAL*8, pointer), `ix` (INTEGER*4), `iy` (REAL*4), `iz` (REAL*8), and `fo` (INTEGER*4). A `Data View` window is open, showing the contents of the `common` block, with `ix` highlighted. Arrows indicate the relationship between the code, the `Call Stack`, and the `Local Variables` panel.

Name	Type	Value
fo	INTEGER*4	2012 (0x000007dc)
xarray	REAL*8, allocatable: (42,11)	(REAL*8, allocatable: (42,11))
fo_o	(Common)	
fo_o	(Common)	
ix	REAL*8, pointer: (42,11)	(REAL*8, pointer: (42,11))
ix	INTEGER*4	1 (0x00000001)
iy	REAL*4	2
iz	REAL*8	3
fo	INTEGER*4	42 (0x0000002a)

For each common block defined in the scope of a subroutine or function, TotalView creates an entry in that function's common block list. The names of common block members have function scope, not global scope. If you select the function in the Call Stack view, the common blocks and their variables appear in the Local Variables panel.

Fortran User-Defined Types

The screenshot illustrates the TotalView interface for Fortran user-defined types. On the left, the Fortran source code for a module is shown. A blue oval highlights the definition of the type `bar`, which contains an integer array `mdarray(2,3,4,5)`. A blue arrow points from this definition to the `just_a_bar` entry in the Data View panel. On the right, the Local Variables panel shows a list of variables, with `just_a_bar` (type `type(bar)`) circled in blue. Another blue arrow points from this entry to the Data View panel. The Data View panel, titled 'Data View', shows a table of variables and their values. The `just_a_bar` entry is expanded, showing its components: `foo_p` (type `type(foo).pointer`), `mdarray` (type `INTEGER*4(...)`), and `foo_array` (type `type(foo)(20)`).

```
module foo_module
  integer dummy           ! Force the module to be defined on cospaq
  type foo
    integer ifoo
  end type foo
  type bar
    integer mdarray(2,3,4,5)
  end type bar
  interface
    subroutine
      integer*4
    end subroutine
    subroutine
      integer*4
    end subroutine
    subroutine
      integer*4
    end subroutine
    subroutine
      integer*4
    end subroutine
  end interface
```

Name	Type	Value
foo_p	type(foo).pointer	(type(foo).pointer::())...
just_a_bar	type(bar)	(type(bar))
mdarray	INTEGER*4(...)	(INTEGER*4(2,3,4,5))
foo_array	type(foo)(20)	(type(foo)(20))
(1)	type(foo)	(Struct)

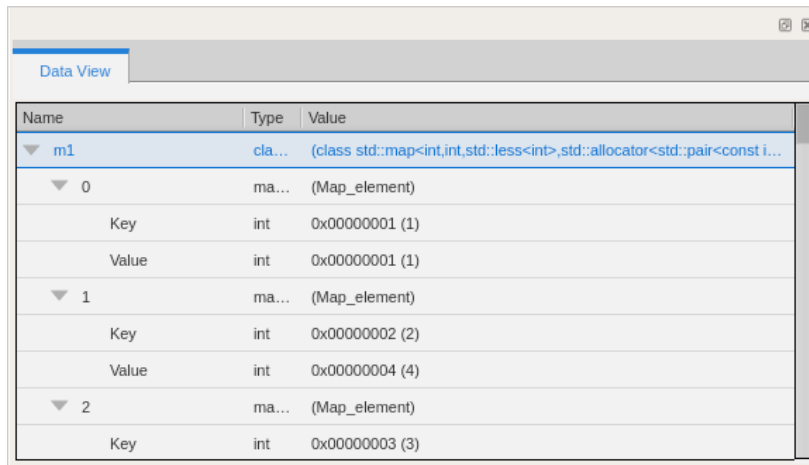
Name	Type
__libc_start_main	
_start	
Local Variables	
chars	character(len=50)
just_a_bar_p	type(bar).pointer
just_a_bar	type(bar)
just_a_foo_p	type(foo).pointer
just_a_foo	type(foo)
foo_array	type(foo)(20)
total	INTEGER*4

TotalView displays user-defined types in the Local Variables panel, which you can then add to the Data View for more detail

Advanced C++ and Data Debugging

C++ Container Transformations

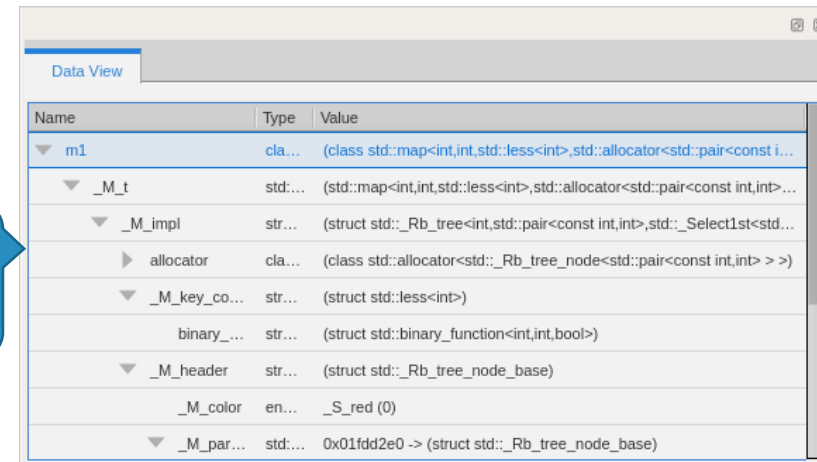
- TotalView transforms many of the C++ and STL containers including:
 - array, forward_list, tuple, map, set, vector and others.



Name	Type	Value
m1	cla...	(class std::map<int,int,std::less<int>,std::allocator<std::pair<const i...
0	ma...	(Map_element)
Key	int	0x00000001 (1)
Value	int	0x00000001 (1)
1	ma...	(Map_element)
Key	int	0x00000002 (2)
Value	int	0x00000004 (4)
2	ma...	(Map_element)
Key	int	0x00000003 (3)

See This!

Instead of This



Name	Type	Value
m1	cla...	(class std::map<int,int,std::less<int>,std::allocator<std::pair<const i...
_M_t	std:...	(std::map<int,int,std::less<int>,std::allocator<std::pair<const int,int>...
_M_impl	str...	(struct std::_Rb_tree<int,std::pair<const int,int>,std::_Select1st<std...
allocator	cla...	(class std::allocator<std::_Rb_tree_node<std::pair<const int,int> > >)
_M_key_co...	str...	(struct std::less<int>)
binary_...	str...	(struct std::binary_function<int,int,bool>)
_M_header	str...	(struct std::_Rb_tree_node_base)
_M_color	en...	_S_red (0)
_M_par...	std:...	0x01fdd2e0 -> (struct std::_Rb_tree_node_base)

Advanced C++ Support

- TotalView supports debugging the latest C++11/14/17 features including:
 - lambdas, transformations for smart pointers, auto types, R-Value references, range-based loops, strongly-typed enums, initializer lists, user defined literals

```
1  #include <functional>
2  #include <vector>
3  #include <iostream>
4  double eval(std::function<double(double)> f, double x = 2.0){
5      return f(x);}
6
7  int main(){
8      // // One line lambdas
9      auto glambda1 = [](int a, float b) { return a < b; };
10     // Two line lambda
11     auto glambda2 = [](int a, float && b) {
12         if (a < b)
13             return 1;
14         if (b>a)
15             return -1;
16         return 0;
17     };
18
19     bool b = glambda1(3, 3.14);
20     int i = glambda2(3, 3.14);
21     for (int i=0; i<10;i++)
22         b = glambda1(i, 3.14+i);
23
24
25     std::function<double(double)> f0    = [](double x){
26         return 1;};
27     auto f1    = [](double x){
28         return x;};
29     decltype(f0) fa[3] = {f0, f1, [](double x){
```

Array Slicing, Striding and Filtering

- Slicing – reduce display to a portion of the array

- [lower_bound:upper_bound]
- [5:10]

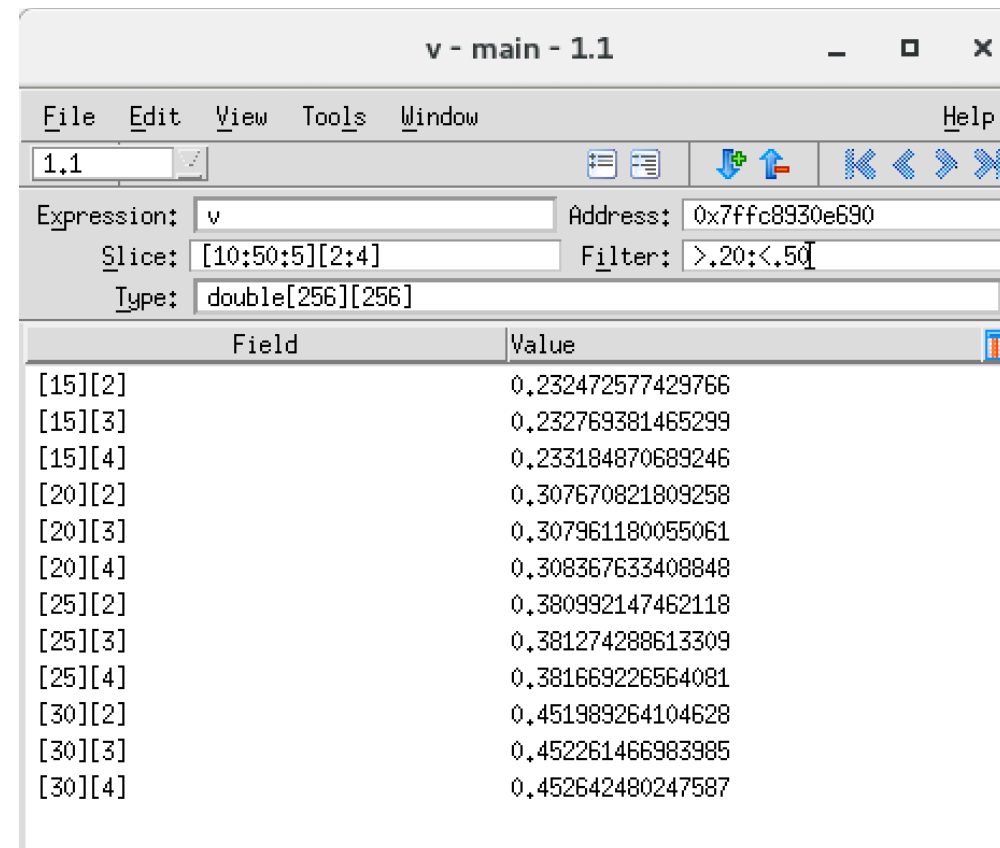
- Striding – Skip over elements

- [::stride]
- [::5], [5:10:-1]

- Filtering

- Comparison: ==, !=, <, <=, >, >=
- Range of values: [>] *low-value* : [<] *high-value*
- IEEE values: \$nan, \$inf, \$denorm

Classic UI Only



Viewing Array Data

- Easily view 2D array data in table format.

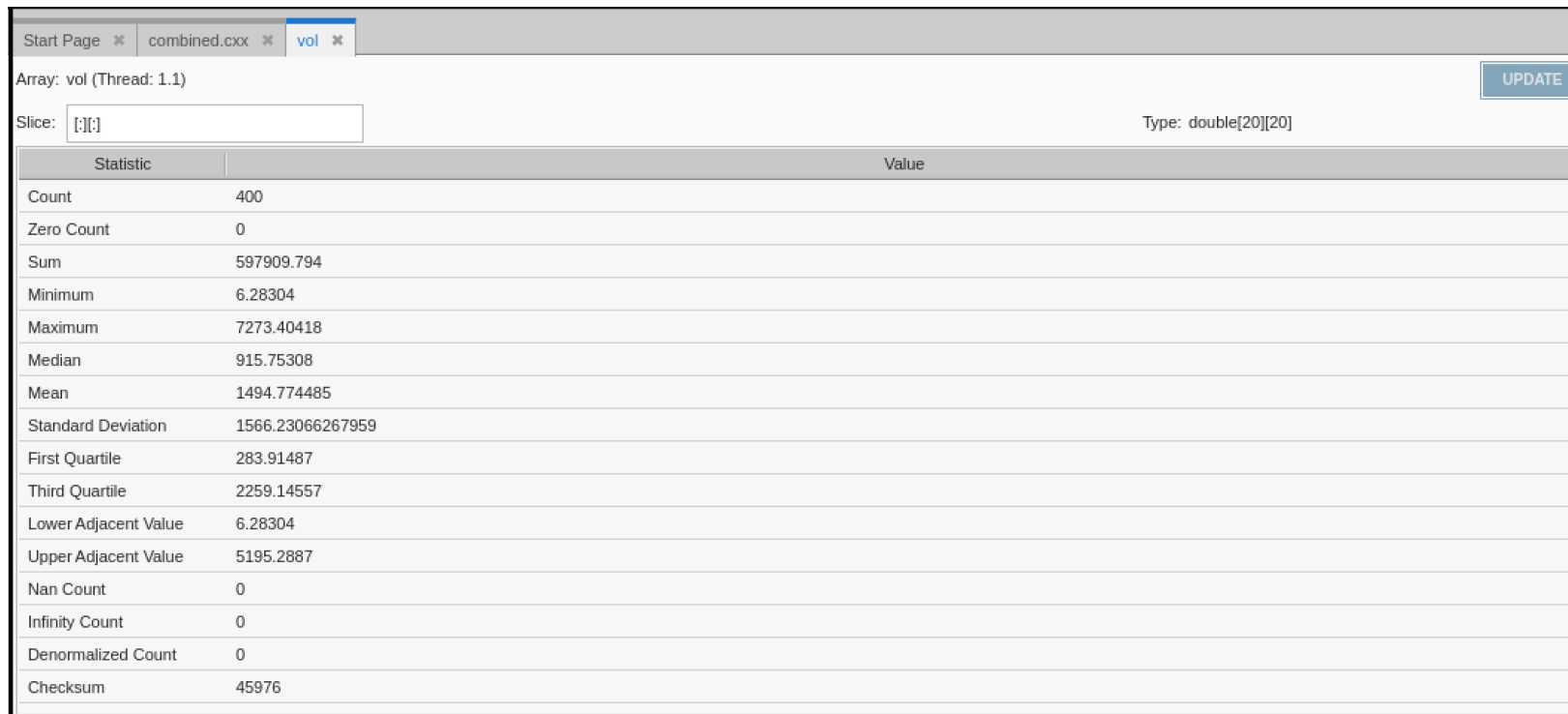
Classic UI Only

The screenshot shows the 'Array Viewer: v[i][j]' window. It has a menu bar with 'File' and 'Help'. Below the menu bar, there are input fields for 'Expression: v' and 'Type: double[256][256]'. A section titled 'Modify array slice:' contains a table with columns 'Dimension', 'Start Index', 'End Index', and 'Stride'. The 'Row' dimension is set from index 10 to 50 with a stride of 5, and the 'Column' dimension is set from index 2 to 4 with a stride of 1. An 'Update View' button is to the right of this table. Below the 'Modify array slice:' section, there is a 'Format: Automatic' dropdown and a 'Slice: [10:50:5][2:4:1]' label. The main area displays a table of array data with columns for the row index, the row slice '[j]:2', and the column slice values. The data is as follows:

	[j]:2	3	4
[i]:10	0.155856154584599	0.156157593783007	0.15657958422173
15	0.232472577429766	0.232769381465299	0.233184870689246
20	0.307670821809258	0.307961180055061	0.308367633408848
25	0.380992147462118	0.381274288613309	0.381669226564081
30	0.451989264104628	0.452261466983985	0.452642480247587
35	0.520229060085251	0.520489664143104	0.520854428381709
40	0.585295244544086	0.585542659988352	0.585888949989877
45	0.646790886958559	0.647023604453132	0.647349307705949
50	0.704340838583089	0.704557438455407	0.704860568035534

Array Statistics

- Easily display a set of statistics for the filtered portion of your array



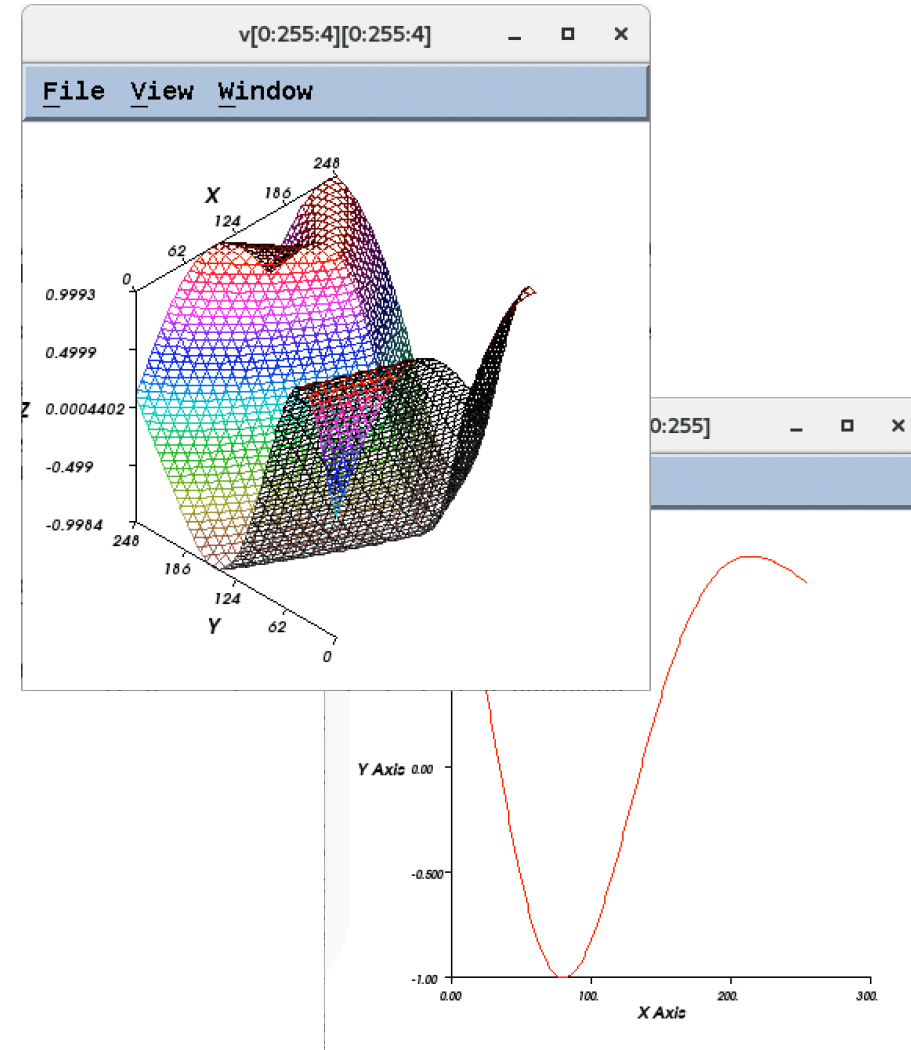
The screenshot shows the 'Array Statistics' window in TotalView. At the top, there are tabs for 'Start Page', 'combined.cxx', and 'vol'. The 'vol' tab is active. Below the tabs, it says 'Array: vol (Thread: 1.1)' and 'Type: double[20][20]'. There is an 'UPDATE' button on the right. A 'Slice:' input field contains '[:]'. Below this is a table with two columns: 'Statistic' and 'Value'.

Statistic	Value
Count	400
Zero Count	0
Sum	597909.794
Minimum	6.28304
Maximum	7273.40418
Median	915.75308
Mean	1494.774485
Standard Deviation	1566.23066267959
First Quartile	283.91487
Third Quartile	2259.14557
Lower Adjacent Value	6.28304
Upper Adjacent Value	5195.2887
Nan Count	0
Infinity Count	0
Denormalized Count	0
Checksum	45976

Visualizing Array Data

- Visualizer creates graphic images of your program's array data.
- Visualize one or two dimensional arrays
- View data manually through the Window > Visualize command on the Data Window
- Visualize data programmatically using the \$visualize function

Classic UI Only



Dive in All

- Dive in All
 - Use Dive in All to easily see each member of a data structure from an array of structures

Data View ✕ Command Line ✕ Logger ✕ Input/Output ✕			
Name	Type	Thread ID	Value
vol		1.1	<Data is Stale>
▶ array	struct compound_t[20]	1.1	(struct compound_t[20])
▼ array[:].x.a	int[20]	1.1	(int[20])
[0]	int	1.1	0x00a700c0 (10944704)
[1]	int	1.1	0x00401853 (4200531)
[2]	int	1.1	0x00405ddd (4218333)
[3]	int	1.1	0x00401720 (4200224)
[4]	int	1.1	0x07b48555 (129271125)
[5]	int	1.1	0x0040180e (4200462)

Q&A

Python Debugging

Python in HPC

- Python development trends:
 - Increased usage of Python to build applications that call out to C++
 - Provides access to
 - High-performance routines
 - Leverage existing algorithms and libraries
 - Utilize advanced multi-threaded capabilities
 - Calling between languages easily enabled using technologies such as **SWIG**, **ctypes**, **Pybind**, Cython, CFFI, etc
 - Debugging mixed language applications is not easy

Python debugging with TotalView

- Debugging one language is difficult enough
- Understanding the flow of execution across language barriers is hard
- Examining and comparing data in both languages is challenging
- What TotalView provides:
 - Easy python debugging session setup
 - Fully integrated Python and C/C++ call stack
 - "Glue" layers between the languages removed
 - Easily examine and compare variables in Python and C++
 - Modest system requirements
 - Utilize reverse debugging and memory debugging
 - Support for Python 2.7 and Python 3.0
- What TotalView does not provide (yet):
 - Setting breakpoints and stepping within Python code

Python without Filtering

The screenshot displays the TotalView debugger interface. The main window shows a C++ source file `tv_python_example.cpp` with the following code:

```
1  /* File: example.c */
2
3  #include "tv_example.h"
4
5  int fact(int n) {
6      if (n < 0) { /* This should probably return an error, but this is simpler */
7          return 0;
8      }
9      if (n == 0) {
10         return 1;
11     }
12     else {
13         /* testing for overflow would be a good idea here */
14         return n * fact(n-1);
15     }
16 }
17
18 int getsquare(int n) {
19     return n * n;
20 }
```

The `fact` function is highlighted, and a breakpoint is set at line 5. The `Call Stack` on the right shows the following frames:

- `fact`
- `_wrap_fact`
- `_PyMethodDef_RawFastCallKeywords`
- `_PyCFunction_FastCallKeywords`
- `call_function`
- `_PyEval_EvalFrameDefault`
- `PyEval_EvalFrameEx`
- `function_code_fastcall`
- `_PyFunction_FastCallKeywords`
- `call_function`

A blue circle highlights the frames from `_wrap_fact` down to `call_function`. A green box with the text **Glue code** is placed over this section of the call stack.

The `Local Variables` pane shows the following arguments:

Name	Type	Value
n	int	0x00000003 (3)

The `Action Points` pane shows two breakpoints:

ID	Type	Stop	Location
1	Break	Group	dot
2	Break	Group	fact

The status bar at the bottom indicates: Process: 1 (2943) python3.7-dbg, Thread: 1.1 (2943) - Breakpoint, Frame: fact, File: ...jects/Python/Python Examples/tv_python_example.cpp, Line: 5.

Python with filtering

The screenshot displays the TotalView debugger interface. The main window shows a C++ source file `tv_python_example.cpp` with the following code:

```
1 /* File: example.c */
2
3 #include "tv_example.h"
4
5 int fact(int n) {
6     if (n < 0) { /* This should probably return an error, but this is simpler */
7         return 0;
8     }
9     if (n == 0) {
10        return 1;
11    }
12    else {
13        /* testing for overflow would be a good idea here */
14        return n * fact(n-1);
15    }
16 }
17
18 int getSquare(int n) {
19     return n * n;
20 }
```

The `fact` function is highlighted in yellow. The `ReplayEngine` button is visible in the toolbar.

The **Processes & Threads** pane on the left shows the following process tree:

Description	# P	# T	Members
python3.7-...	1	1	p1
Br...	1	1	p1
fact	1	1	p1.1
	1	1	p1.1

The **Call Stack** pane on the right shows the following stack:

Language	Function
C++	fact
Py	callFact
Py	pySupportedTypes
Py	<module>
C	main
	__libc_start_main

The **Data View** pane at the bottom shows the following variables:

Name	Type	Thread ID	Value
[Add New Expression]			

The **Local Variables** pane on the right shows the following variables:

Name	Type	Value
Arguments		
n	int	0x00000003 (3)

The **Action Points** pane at the bottom left shows the following breakpoints:

ID	Type	Stop	Location
1	Break	Group	dot
2	Break	Group	fact

The status bar at the bottom indicates: Process: 1 (2943) python3.7-dbg, Thread: 1.1 (2943) - Breakpoint, Frame: fact, File: ...jects/Python/Python Examples/tv_python_example.cpp, Line: 5.

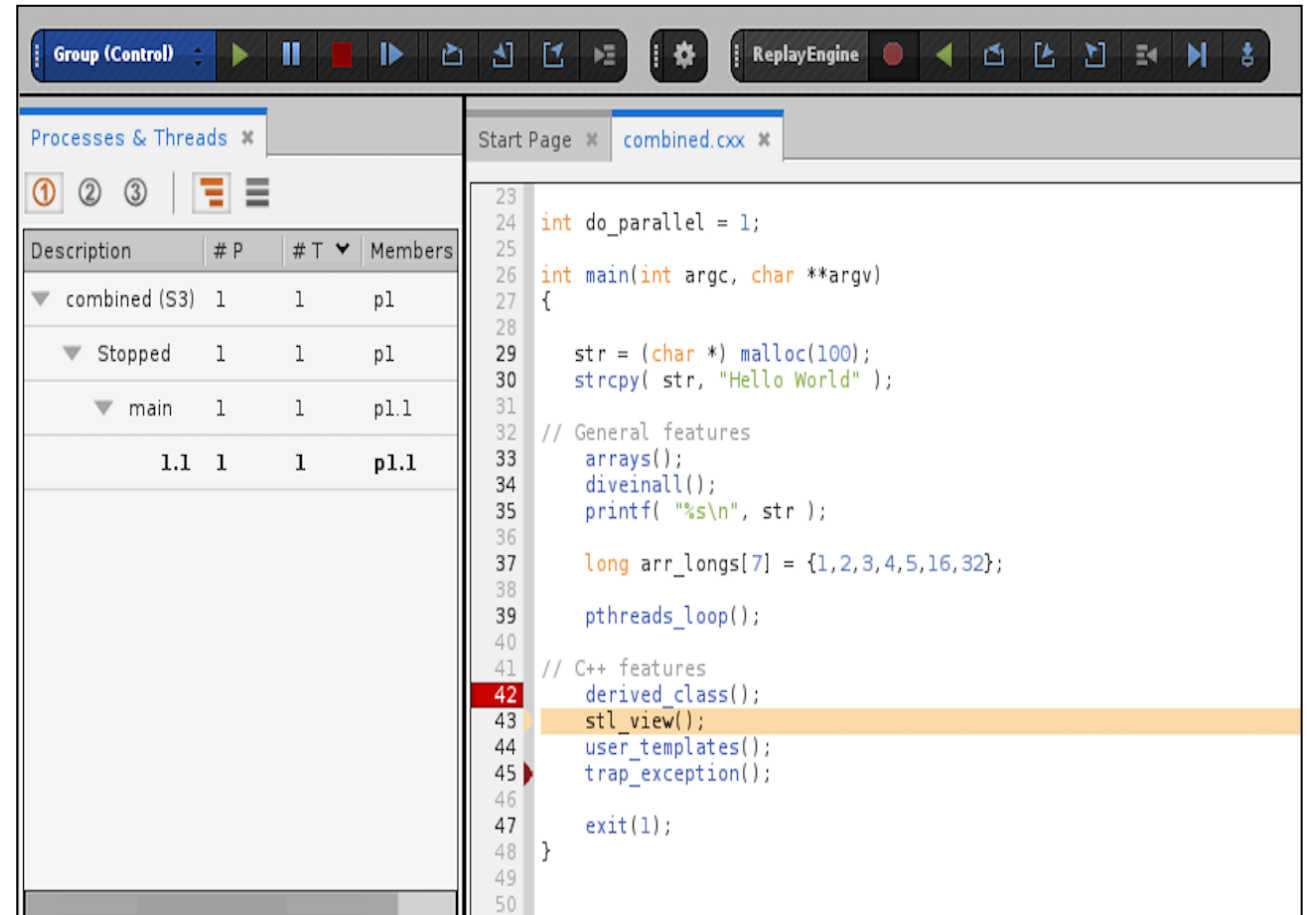
Demo

- TotalView Python / C++ debugging demo (test_python_types.py)

Reverse Debugging with Replay Engine

Reverse debugging

- How do you isolate an intermittent failure?
 - Without TotalView
 - Set a breakpoint in code
 - Realize you ran past the problem
 - [Re-load](#)
 - Set breakpoint earlier
 - [Hope it fails](#)
 - [Keep repeating](#)
 - With TotalView
 - Set a breakpoint
 - Start recording
 - See failure
 - Run backwards/forwards in context of failing execution
 - Reverse Debugging
 - Re-creates the context when going backwards
 - Focus down to a specific problem area easily
 - Saves days in recreating a failure





Recording and Playback


- When ReplayEngine is saving state information, it is in **Record Mode**
- The saved state information is the program's execution history
- You can save the execution history at any time and reload the recording when debugging the executable in a subsequent session
- Using a ReplayEngine command, ether from the Toolbar or the CLI, shifts ReplayEngine into **ReplayMode**
- Debugging commands that do not work in ReplayMode include:
 - Changing a variable's value
 - Functions that alter memory
 - Running threads asynchronously


ReplayEngine Toolbar




Record  a toggle that enables and disables ReplayEngine.


Go Back  displays the state that existed at the last action point. If no action point is encountered, ReplayEngine displays the state that existed at the start of its recorded history.

Prev  displays the state that existed when the previous statement executed. If that line had a function call, Prev skips over the call.

Unstep  displays the state that existed when the previous statement executed. If that line had a function call, ReplayEngine moves to the last statement in that function.

Caller  displays the state that existed before the current routine was called.

Back To  displays the program's state for the line you select. This line must have executed prior to the currently displayed line.

Live  shifts from replay mode to record mode. It also displays the statement that would have executed had you not moved into ReplayMode.

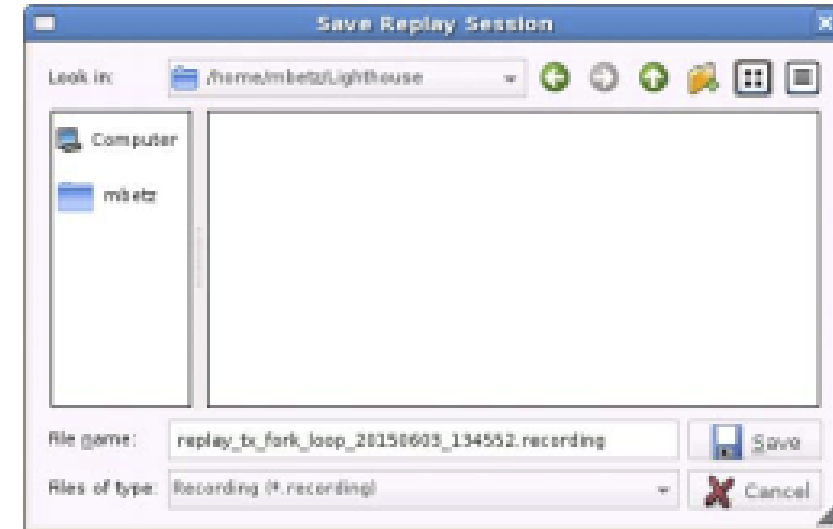
Bookmark  creates a ReplayEngine bookmark at a selected location.

Save  saves the current replay recording session to a file.

Saving and Loading Execution History

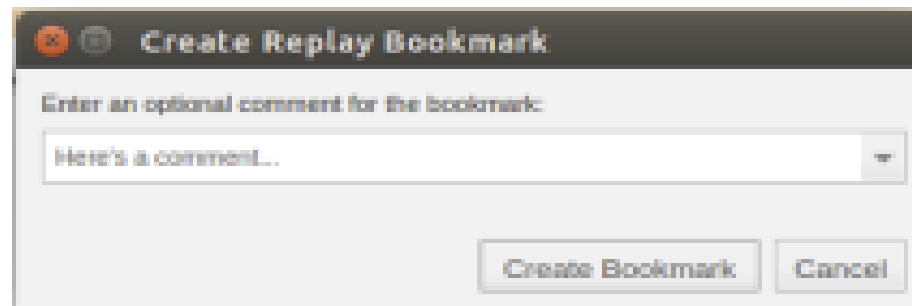
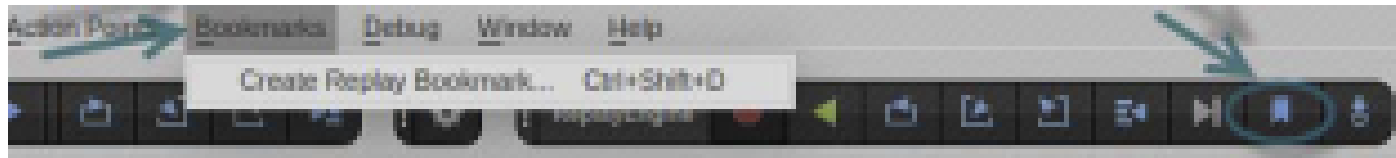
- TotalView can save the current ReplayEngine execution history to file at any time
- The saved recording can be loaded into TotalView using any of the following:
 - At startup, using the same syntax as when opening a core file:

`totalview -newUI executable recording-file`
 - On the Start Page view by selecting Load Core File or Replay Recording File

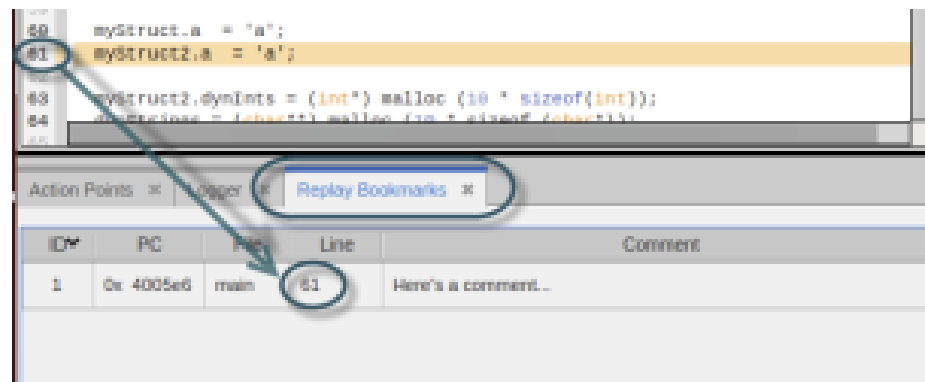


Replay Bookmarks

- Replay bookmarks mark a point in the execution of a program, allowing you to quickly jump back to that point in time



Creating a Replay Bookmark



Activating a Replay Bookmark

Setting Preferences for ReplayEngine

- You can set the following preferences for ReplayEngine
 - the maximum amount of memory to allocate to ReplayEngine
 - The preferred behaviour when the memory limit is reached
- Setting the maximum amount of memory. The default value '0' specifies to limit the maximum size by available memory only.

`dset TV::replay_history_size value`

e.g. `dset TV::replay_history_size 1024M`

- Setting the preferred behaviour. By default, the oldest history is discarded so that recording can continue

`dset TV::replay_history_mode 1` (Discard oldest history and continue recording)

`dset TV::replay_history_mode 2` (Stop the process when the buffer is full)

Demo

- TotalView ReplayEngine Demo

Debugging OpenMP Applications

TotalView support for OpenMP

Some of the features TotalView supports:

- Source-level debugging of the original OpenMP code
- The ability to plant breakpoints throughout the OpenMP code, including lines that are executed in parallel
- Visibility of OpenMP worker threads
- Access to SHARED and PRIVATE variables in OpenMP PARALLEL code
- Access to OMP THREADPRIVATE data in code compiled by supported compilers

About TotalView OpenMP Features

- The compiler can generate multiple outlined routines from a single parallel region. This means that a single line of source code can generate multiple blocks of machine code inside different functions
- You can't single step into or out of a parallel region. Instead, set a breakpoint inside the parallel region and let the process run to it. After execution reaches the parallel region, you can single step in it
- OpenMP programs are multi-threaded programs, so the rules for debugging multi-threaded programs apply

Demo

- TotalView OpenMP demo (tx_omp_c_threadprivate)

Debugging Parallel Applications

Multi-Thread and Multi-Process Debugging

- TotalView provides the power to
 - Simultaneously debug many threads and processes in a single debugging session
 - Supports MPI, fork/exec, OpenMP, pthreads, std::thread, et al
 - Help locate deadlocks and race conditions
 - Understand complex applications utilizing threads
- By
 - Providing control of entire groups of processes, individual processes or even down to individual threads within a process
 - Enabling thread level breakpoints and barrier controls
 - Showing aggregated thread and process state display

Starting a Parallel Program Session from the UI

- From New Parallel Session page select:
 - MPI preference
 - Number of tasks
 - Number of nodes
 - Starter arguments
- Click Start Session to save and launch

The screenshot shows the 'Session Editor' window with a close button (X) in the top right corner. The window is divided into two main columns. The left column contains three sections: 'Session Details' with a 'Session Name' dropdown menu set to 'mpi_array'; 'Parallel Details' with a 'Parallel System' dropdown menu set to 'Open MPI' (marked as 'REQUIRED'); and 'Tasks (-np):' with a text input field containing '4'. Below these is a text area for 'Additional Starter Arguments' with the placeholder '[Enter starter arguments as needed]'. The right column contains three sections: 'Program Details' with a 'File Name' dropdown menu set to 'Projects/LLNL_MPI_Examples/LLNLMPIDExamples/mpi_array' (marked as 'REQUIRED') and a 'BROWSE...' button; 'Arguments' with a text area containing '[Enter any program arguments. Ex. -option foo]'; and 'Debug Options' with checkboxes for 'Reverse Debugging' and 'Python Debugging', both of which are unchecked. Below these is a section for 'Program Environment' with a text area for 'Environment variables for the program' containing the placeholder '[Enter line-separated NAME=VALUE pairs]'. At the bottom right of the window are three buttons: 'RESET', 'LOAD SESSION', and 'CANCEL'.

Session Editor

Parallel Session

Session Details

Session Name

mpi_array

Parallel Details

Parallel System

Open MPI

Tasks (-np):

4

Additional Starter Arguments

[Enter starter arguments as needed]

Standard Input Redirection

Standard Output/Error Redirection

Program Details

File Name

Projects/LLNL_MPI_Examples/LLNLMPIDExamples/mpi_array

BROWSE...

Arguments

[Enter any program arguments. Ex. -option foo]

Debug Options

Reverse Debugging

☐ Enable reverse debugging with ReplayEngine

Python Debugging

☐ Enable call stack filtering for Python

Program Environment

Environment variables for the program

[Enter line-separated NAME=VALUE pairs]

RESET LOAD SESSION CANCEL

Starting a Parallel Program Session from the Command Line

MPI	Startup Command
IBM	<code>totalview --args poe myprog -procs 4 -rmpool 0</code>
QUADRICS Intel Linux under SLURM	<code>totalview --args srun -n 16 -p pdebug myprog</code>
MVAPICH Opteron Linux under SLURM	<code>totalview --args srun -n 16 -p pdebug myprog</code>
SGI	<code>totalview --args mpirun myprog -np 16</code>
Sun	<code>totalview --args mprun myprog -np 16</code>
MPICH	<code>mpirun -np 16 -tv myprog</code>
MPICH2 Intel MPI	<code>totalview --args python 'which mpiexec' -tv su -np 16 myprog</code>

The order of arguments and executables differs between platforms

You can use `totalview --args` instead of `totalview <starter> -a`

Parallel Debugging Group, Process and Thread Control

The screenshot displays the TotalView debugger interface. On the left, the 'Group (Control)' menu is open, showing options for 'Group (Control)', 'Process', and 'Thread'. A blue arrow points from the text 'Select either' to this menu. Below the menu is a table listing debug targets:

Description	# P	# T	Members
mpirun (S3)	1	1	p1
Running	1	1	p1
1	1	1	p1
demoMpi_v2 (S4)	4	4	0-3
Breakpoint	4	4	0-3
2	1	1	0
3	1	1	1
4	1	1	2
5	1	1	3

The central pane shows the source code of `demoMpi_v2.C`. Line 72 is highlighted in yellow, containing the code: `if(myid%2 == 0) sleep(2);`. The right pane shows the 'Call Stack' with entries for `main`, `_libc_start_main`, and `_start`. The bottom pane shows the 'Local Variables' section with variables `argc` (int, 0x00000001), `argv` (string **, 0x7ffe60fba08), and `root` (int, 0x00000000).

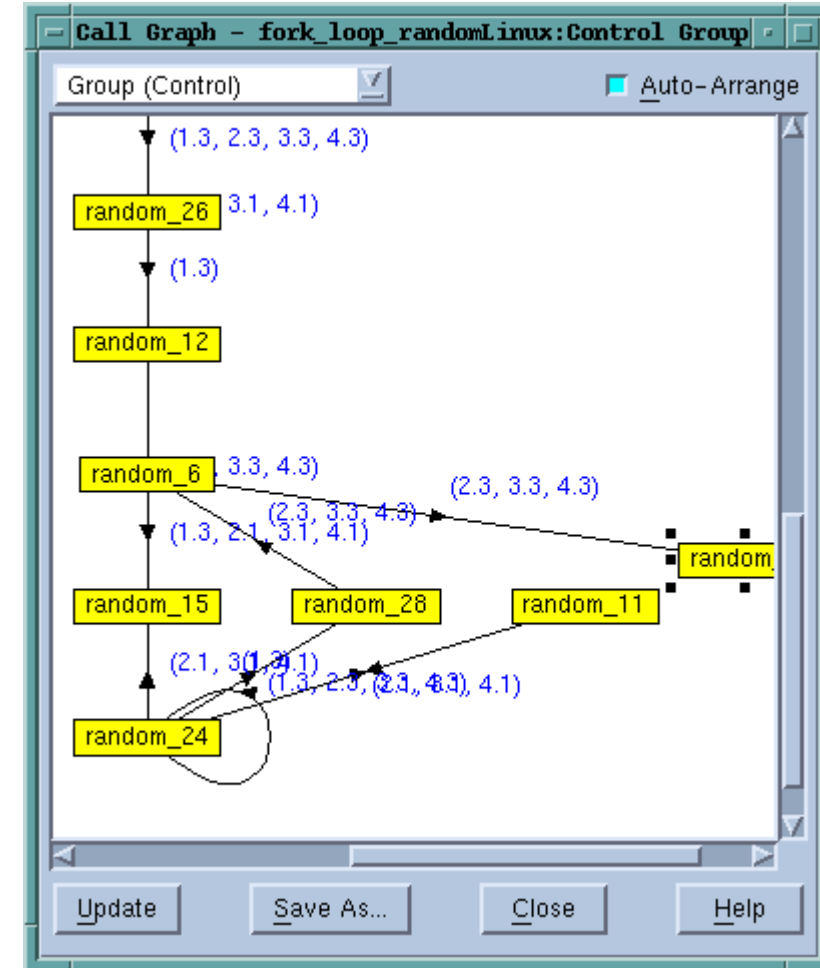
Select either

- Group
- Process
- Thread

Call Graph

Classic UI Only

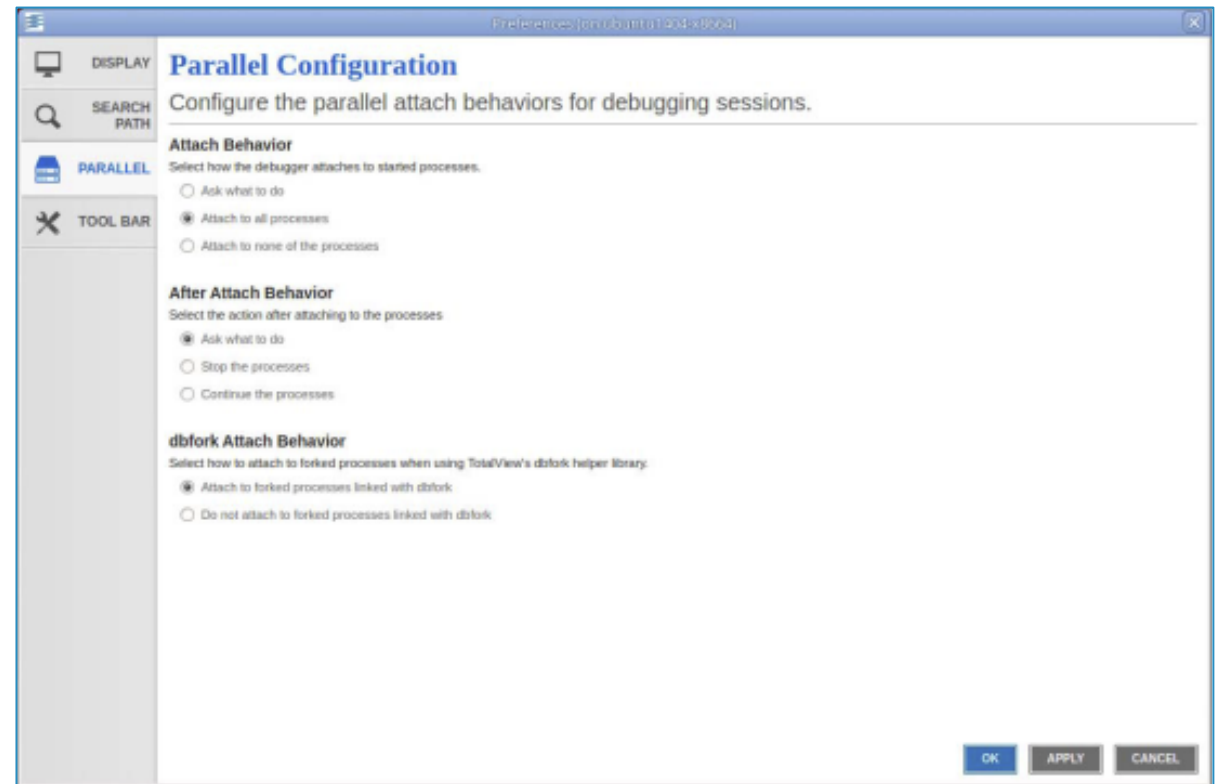
- Quick view of program state
 - Each call stack is a path
 - Functions are nodes
 - Calls are edges
 - Labeled with the MPI rank or thread ID
 - Construct process groups
 - Look for outliers
- Dive on a node in the call graph to create a Call Graph group.



Parallel Preferences

Attach Behavior controls if TotalView should attach to all of the processes, none or ask what to do

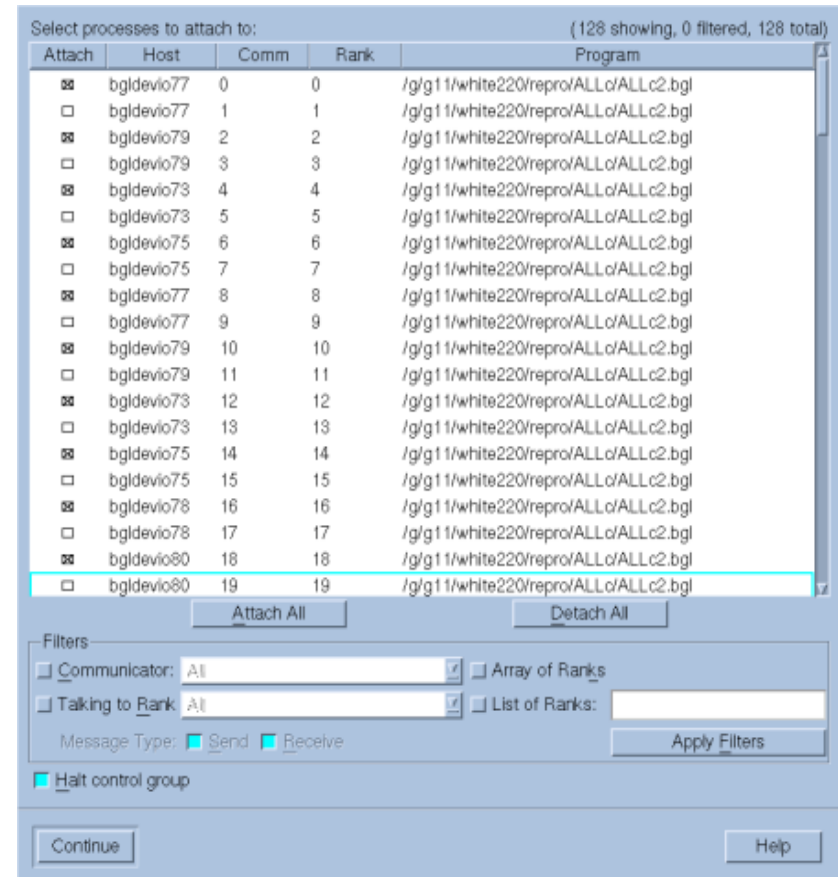
After Attach Behavior controls if parallel job stops, runs or if TotalView should ask what to do



Subset Attach - Control Which Processes TotalView Attaches To

Classic UI Only

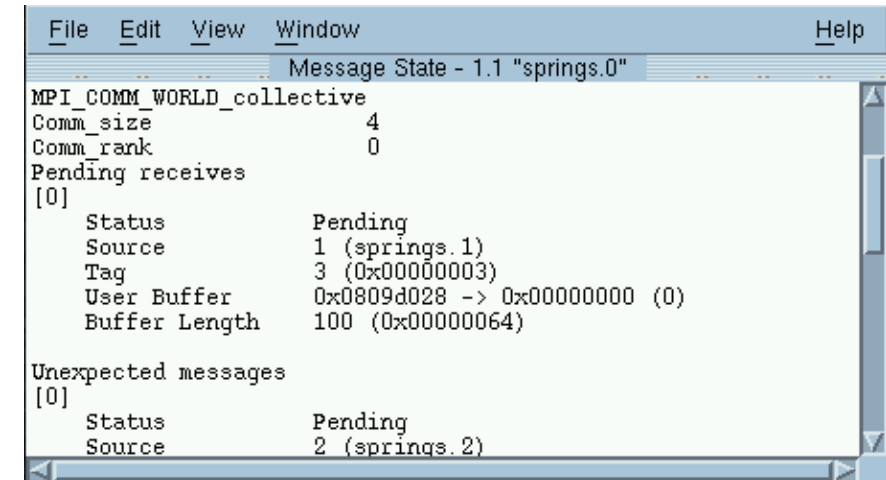
- Debug a subset of the processes that make up the job
 - Sometimes the user does not need to control and see every process to understand the behavior or id the defect
- The subset can be changed at any time
 - Can narrow, expand or shift focus
- Uncouples interactive performance from job size
 - After the subset operation completes
 - Interactive performance depends on subset size
- Supports the use of lightweight tools
 - Such as LLNL's STAT



View MPI Message Queues

- Information visible whenever MPI rank processes are halted
- Provides information from the MPI layer
 - Unexpected messages
 - Pending Sends
 - Pending Receives
- Use this info to debug
 - Deadlock situations
 - Load balancing
- May need to be enabled in the MPI library
 - --enable-debug

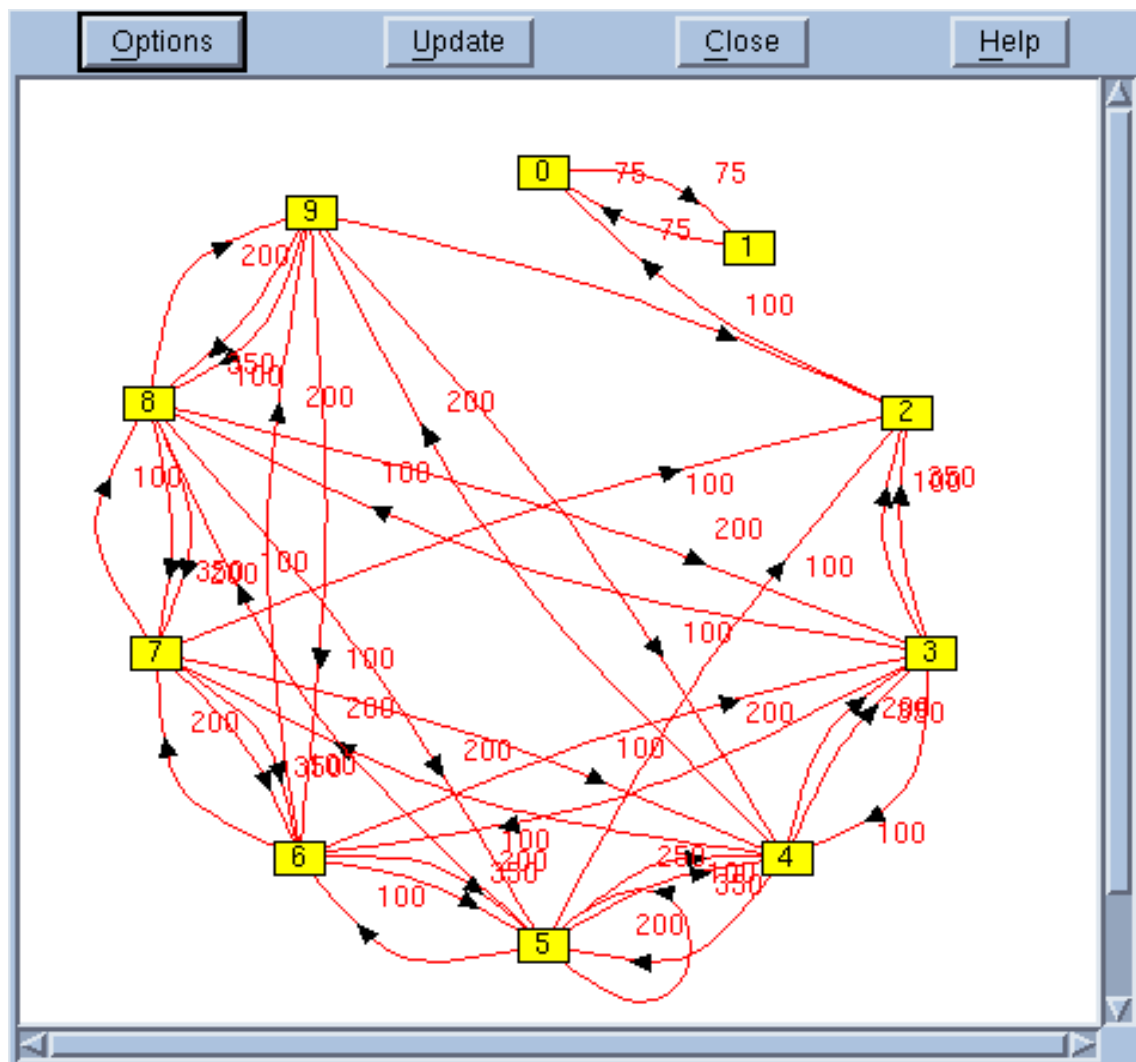
Classic UI Only



Message Queue Graph

Classic UI Only

- Hangs & Deadlocks
 - Receives
 - Sends
 - Unexpected
- Inspect
 - Individual entries
- Patterns

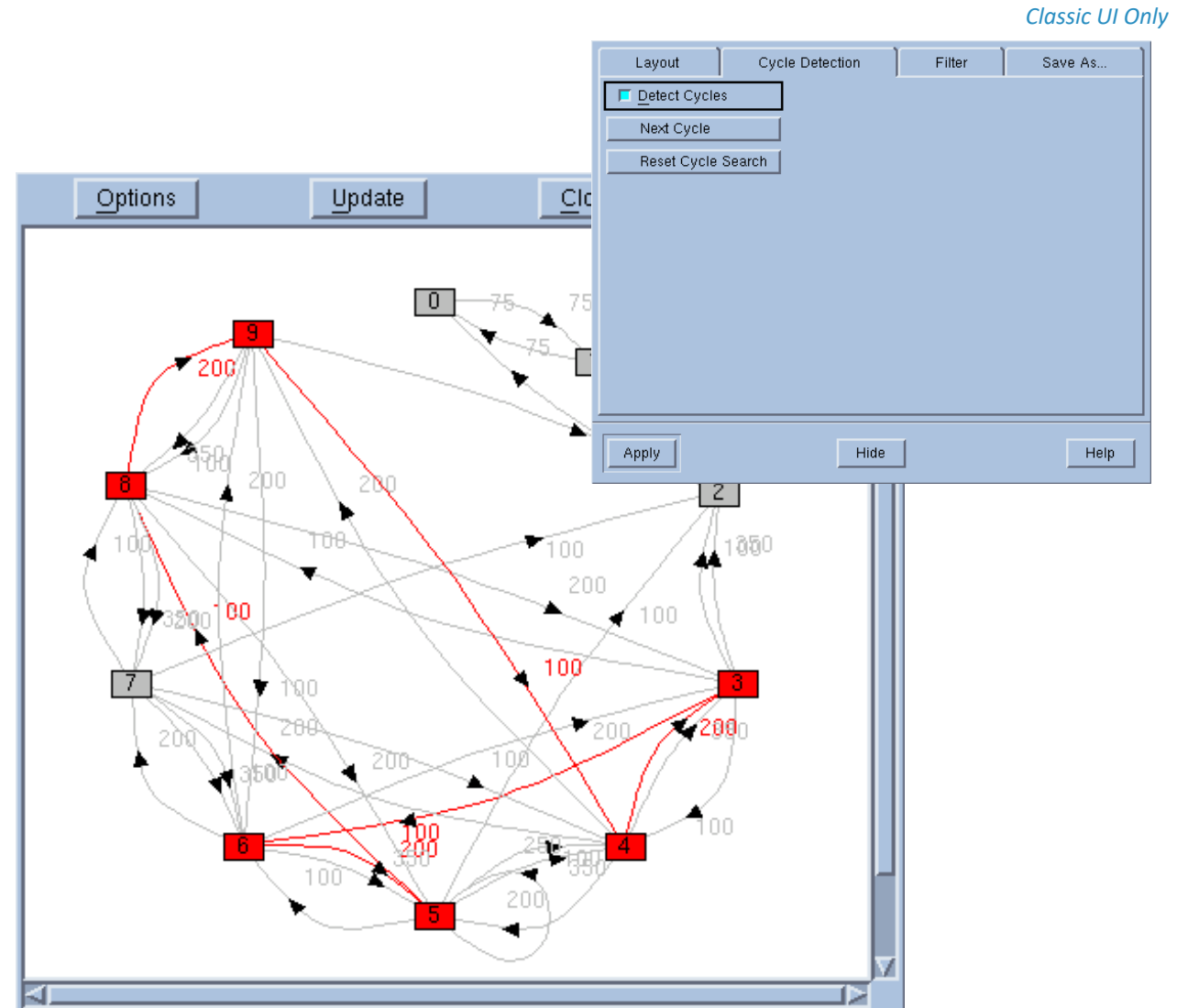
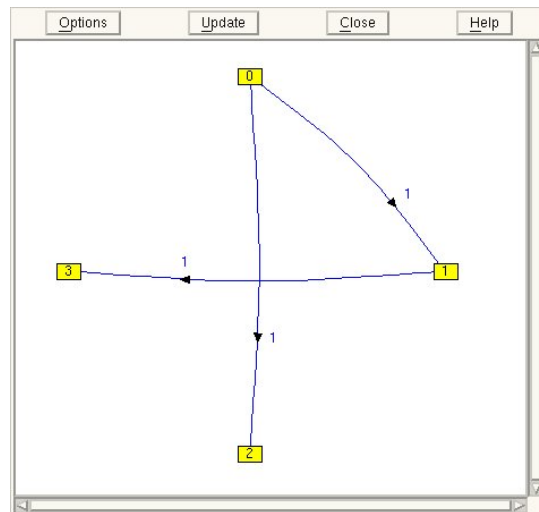


Find Deadlocks and Performance Sinks

- Filtering

- Choose messages to track
- Choose MPI Communicators

- Cycle detection



Multi-Thread Debugging Techniques

- Multiple ID's for threads
 - pthread library ID – Displayed by default in TotalView
 - OS Light Weight Process (LWP) ID
 - TotalView thread ID – ProcessID.ThreadID, e.g. 1.3
- Finding deadlocks due to mutex misuse
 - Utilize ReplayEngine/reverse debugging
 - Leverage watchpoints to find when mutex was acquired
 - Set the “Open process window at breakpoint” preference on the Action Points tab
 - To get LWP id, turn off TotalView user threads (-no_user_threads)
 - TotalView normally just displays the pthread ID

Multi-Thread Debugging Techniques

- Dealing with thread starvation
 - A tough problem to solve...
 - Utilize prior technique for watching when mutex's are locked/unlocked
 - Leverage Evaluation Points and TotalView's built-in Statements
 - \$countthread expression
 - \$holdthread
 - \$stopthread
 - Halt the program during execution several times to see where execution is at in the Stack Trace

Multi-Process Debugging Techniques

- For high-scale debugging sessions, use command line launch of the parallel job instead of the Parallel Program Session in UI.
 - UI Parallel Program Session uses a flexible “bootstrap” parallel session mechanism for easy debug session setup but takes longer to launch.
- Enable reverse debugging on a per-process basis
 - Halt a specific process and enable reverse debugging on the fly
- Memory debugging can be enabled on one or more processes

Demo

- TotalView MPI Demo (mpi_array_broken)

Memory Debugging with MemoryScape

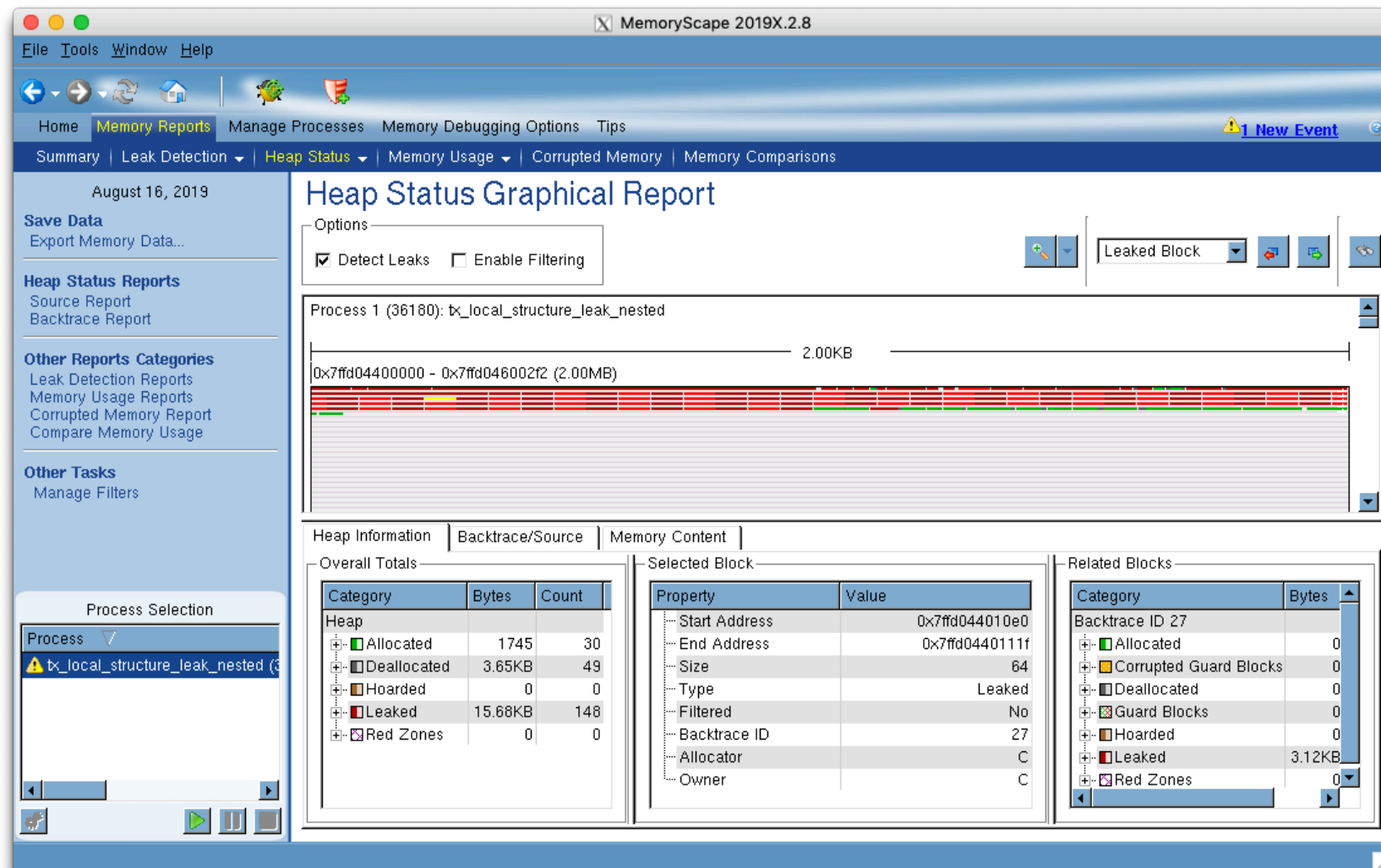
MemoryScape – Simple to use, intuitive memory debugging

- What is MemoryScape?

- Streamlined
- Lightweight
- Intuitive
- Collaborative
- Memory Debugging

- Features

- Shows
 - Memory errors
 - Memory status
 - Memory leaks
 - Buffer overflows
- MPI memory debugging
- Remote memory debugging



What is a Memory Bug?

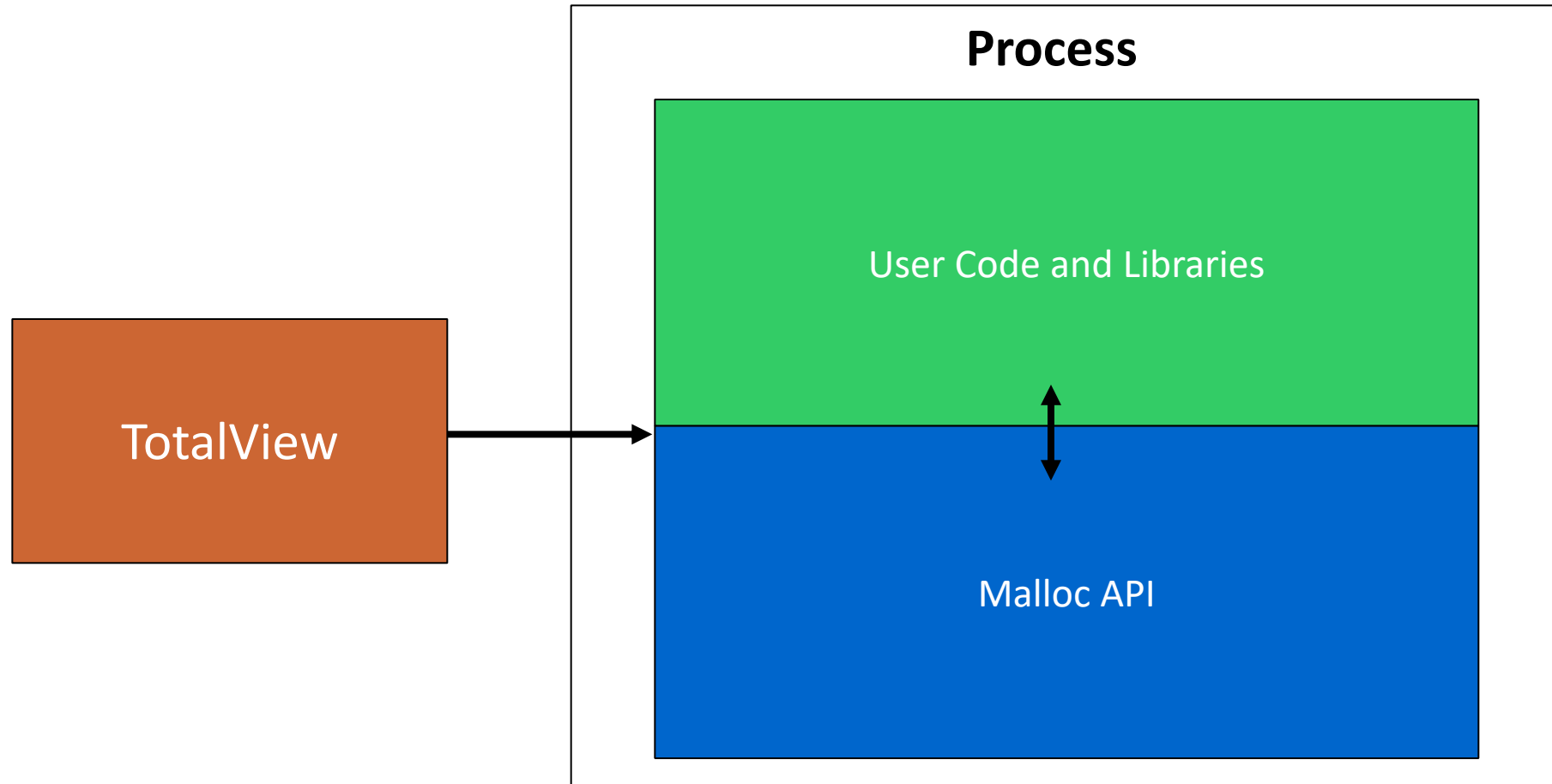
- A Memory Bug is a mistake in the management of heap memory
 - Leaking: Failure to free memory
 - Dangling references: Failure to clear pointers
 - Failure to check for error conditions
 - Memory Corruption
 - Writing to memory not allocated
 - Overrunning array bounds

Heap Memory

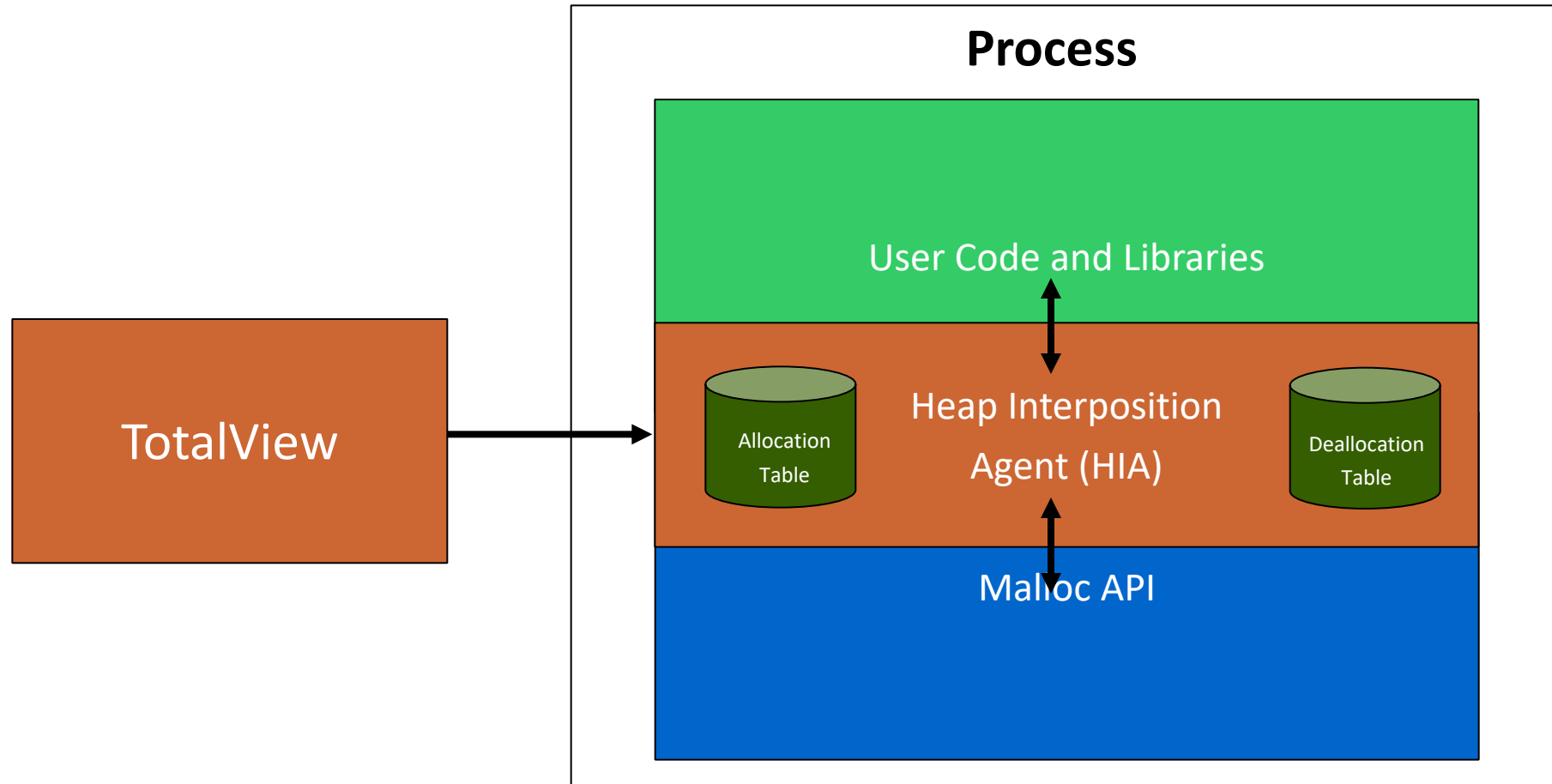
- Heap is managed by the program
 - C: Malloc() and free()
 - C++: New and Delete
 - Fortran90: Allocatable arrays
- Malloc usage is something like:

```
int * vp;  
vp=malloc(sizeof(int)*number);  
if (vp == 0){ /* malloc must have failed*/ }  
/* use vp */  
free(vp);  
vp=0;
```

The Agent and Interposition



The Agent and Interposition



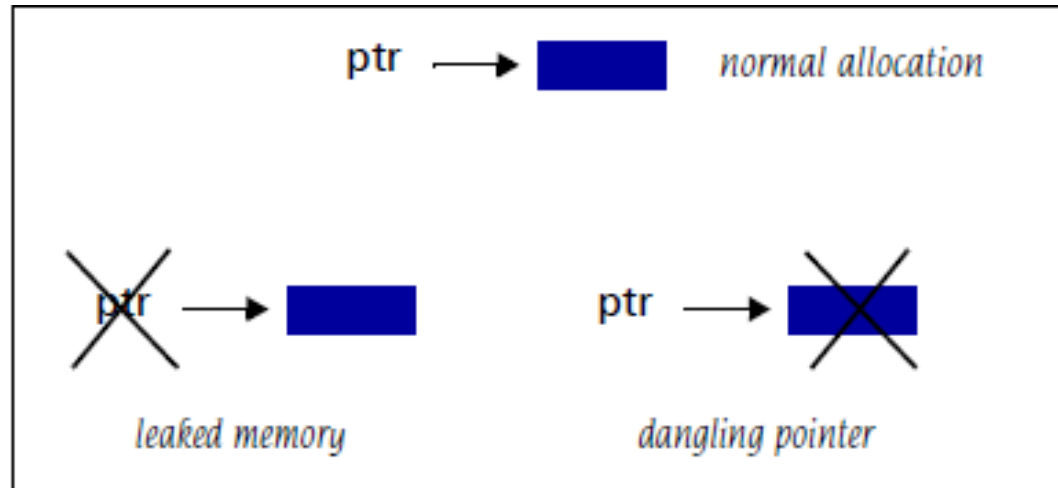
TotalView HIA Technology

- Advantages of TotalView HIA Technology
 - Use it with your existing builds
 - No Source Code or Binary Instrumentation
 - Programs run nearly full speed
 - Low performance overhead
 - Low memory overhead
 - Efficient memory usage

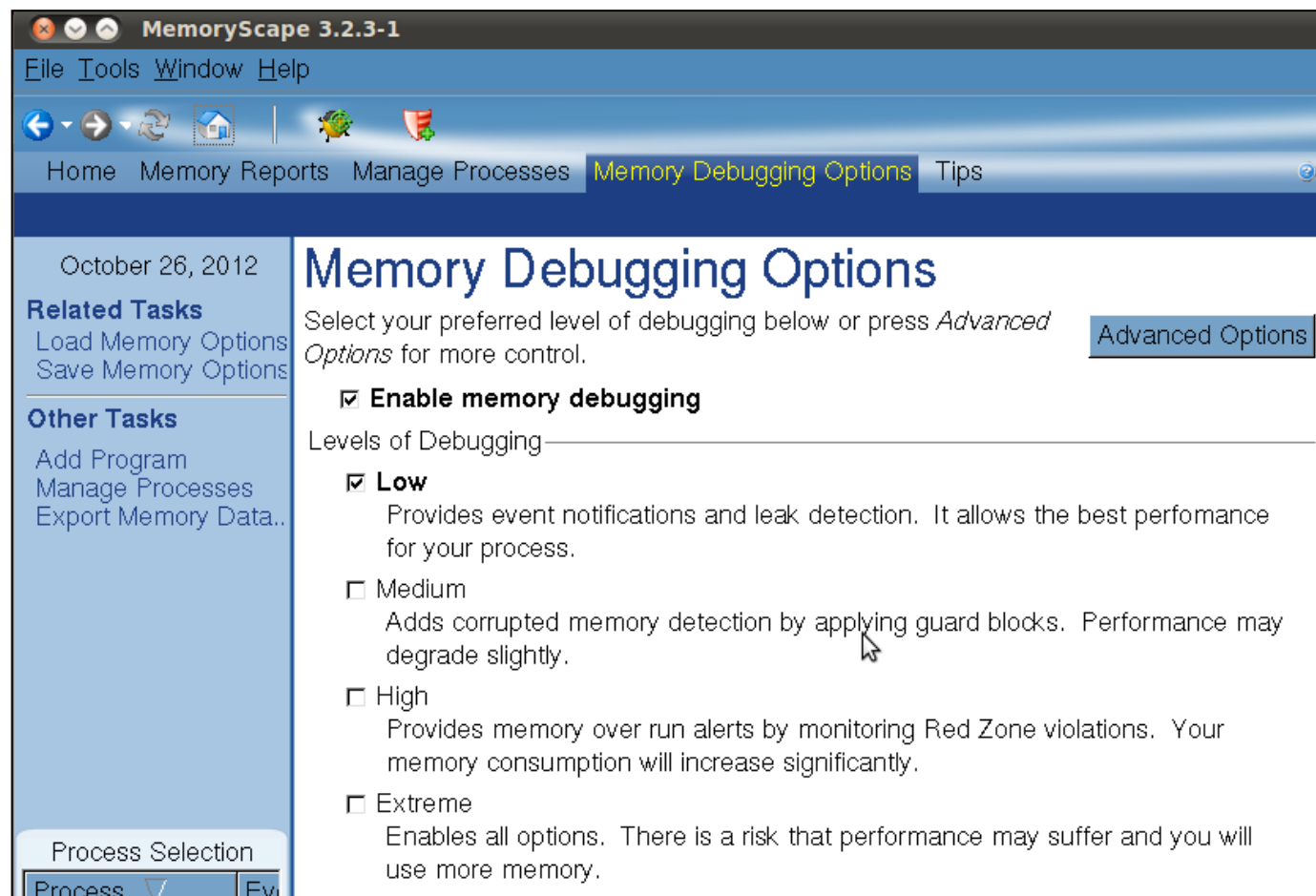
Memory Debugger Features

- Automatically detect allocation problems
- Memory Corruption Detection - Guard Blocks & Red Zones
- Leak detection
- Dangling pointer detection
- View the heap
- Memory Hoarding
- Memory Comparisons between processes

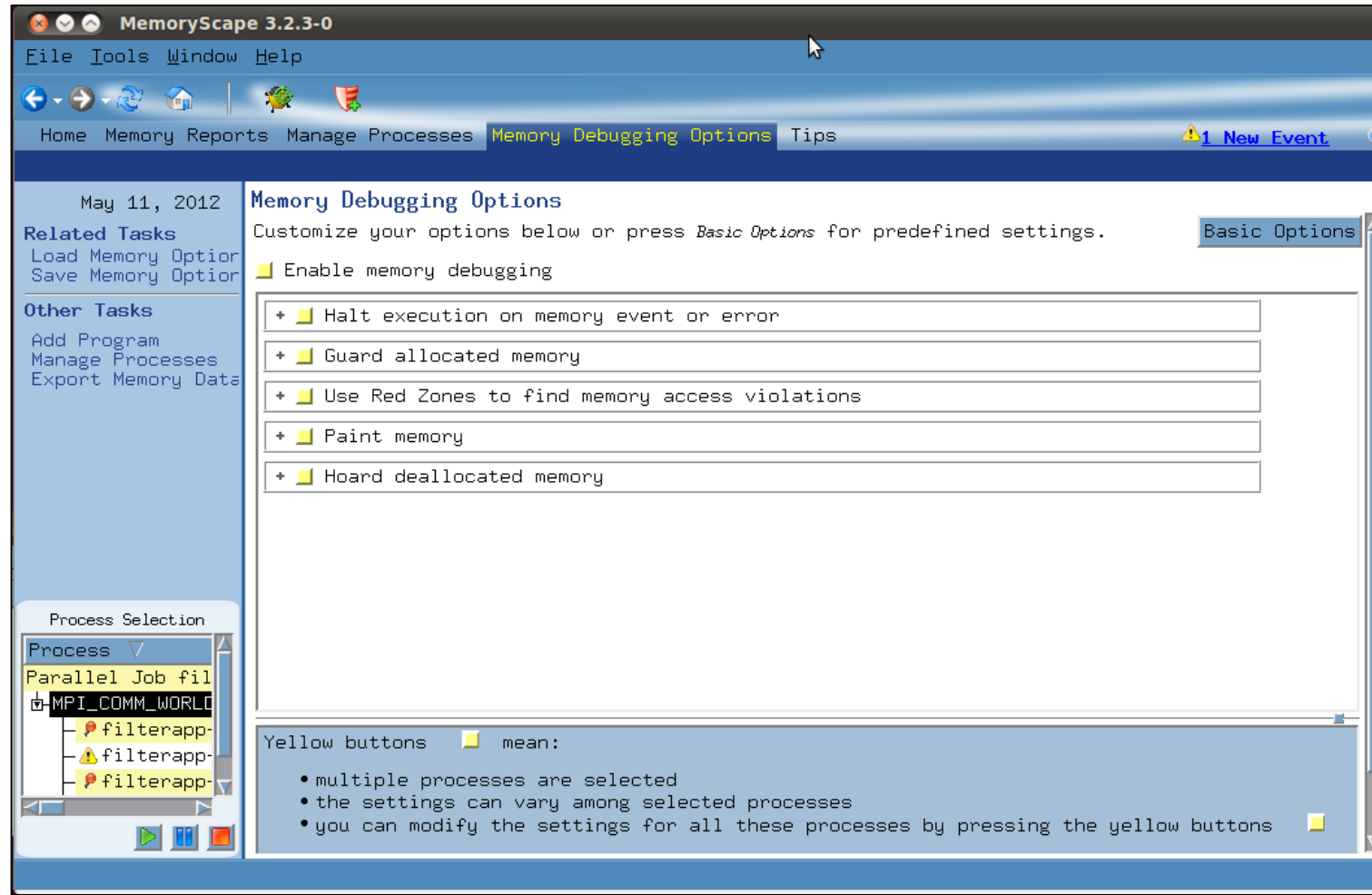
Leaks and Dangling Pointers



Memory Debugging Options



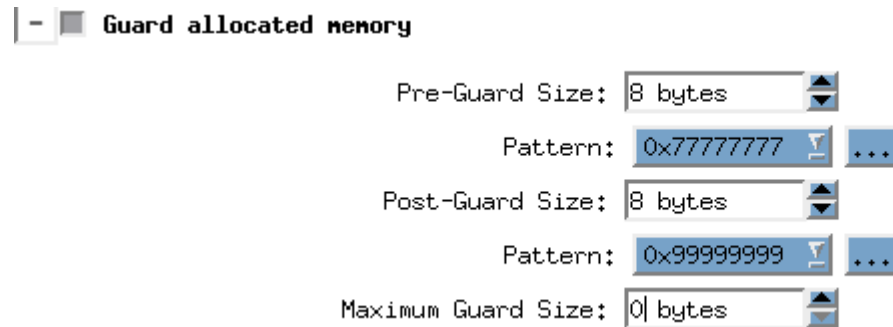
Advanced Memory Debugging Options



Guard Blocks

Guard allocated memory

When selected, the Memory Debugger writes guard blocks before and after a memory block that your program allocates



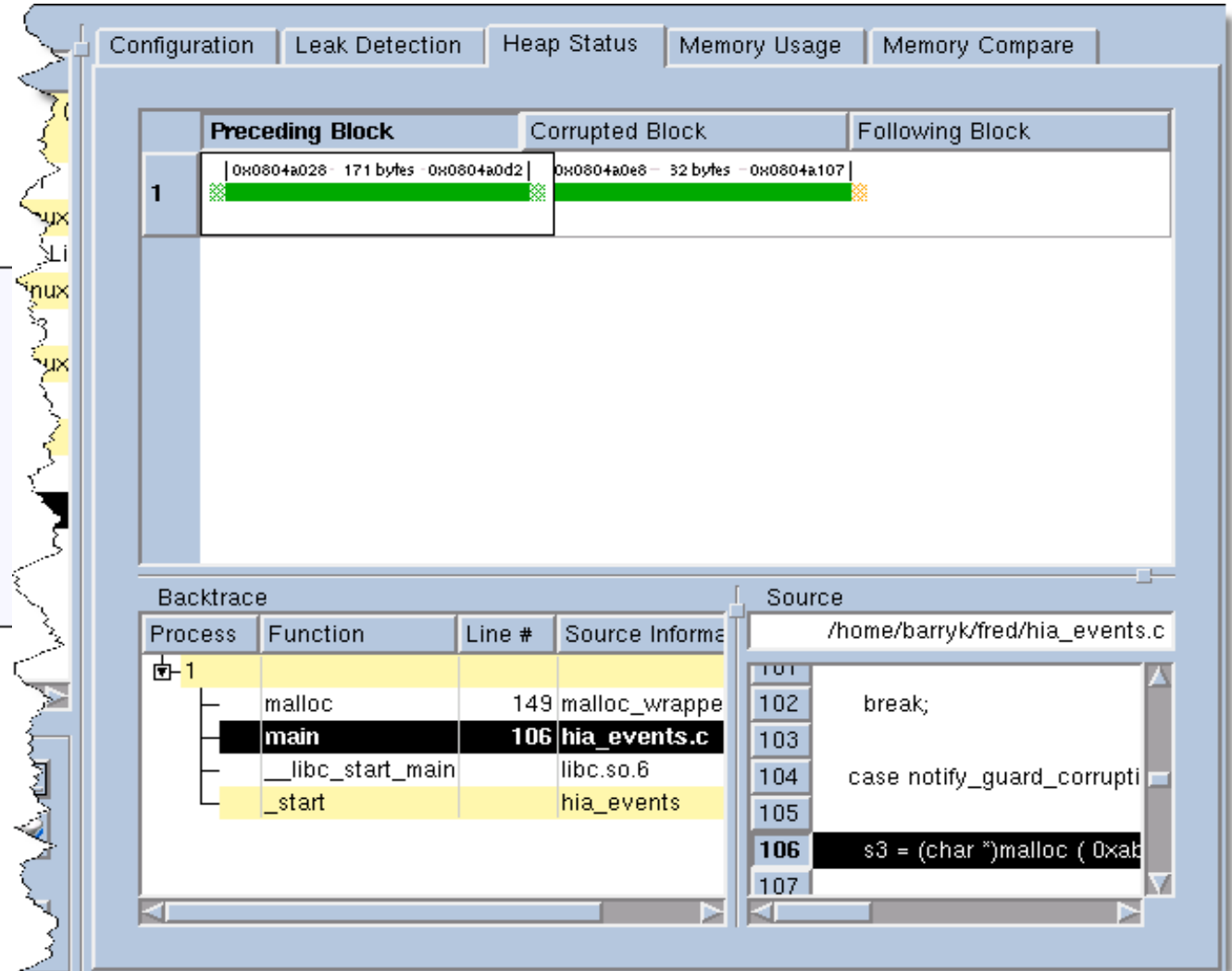
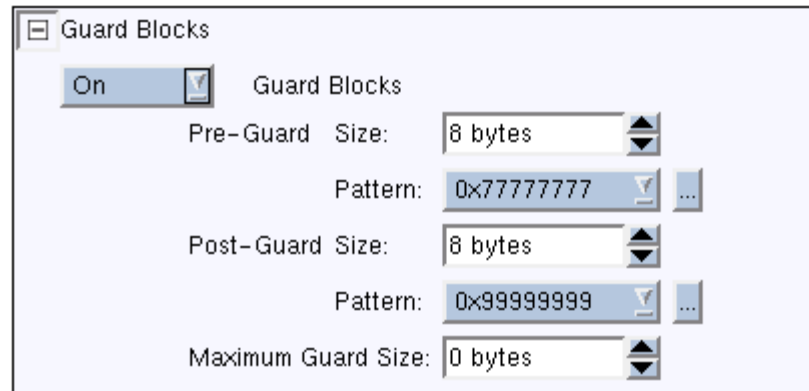
Pre-Guard and Post-Guard Size:

Sets the size in bytes of the block that the Memory Debugger places immediately before and after the memory block that your program allocates

Pattern:

Indicates the pattern that the Memory Debugger writes into guard blocks. The default values are 0x77777777 and 0x99999999

Guard Blocks



Red Zones

Red Zones : instant array bounds detection

- Red Zones provides:
 - Both read and write memory access violations.
 - Immediate detection of memory overruns.
 - Detection of access violations both before and after the bounds of allocated memory.
 - Detection of deallocated memory accesses.
- Red Zones events
 - MemoryScape will stop your programs execution and raise an event alerting you to the illegal access. You will be able to see exactly where your code over stepped the bounds.

Red Zones

Red Zones instant array bounds detection for Linux

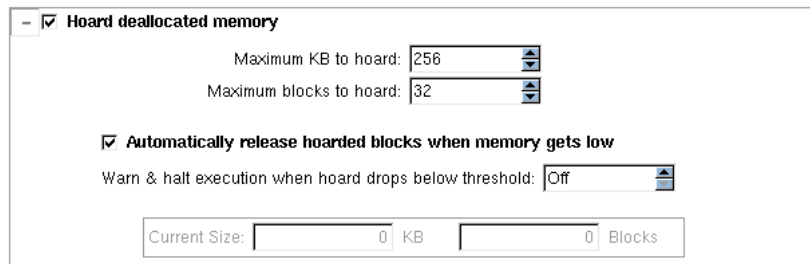
- Red Zones allocation size range controls
 - The optional use of Red zones will increase the memory consumption of your program.
 - Controls are provided to allow the full management of Red Zone usage. These controls allow:
 - Restriction of red zones to allocations in several user defined size ranges
 - Easily turning red zones on and off at any time during your programs execution.

Memory Painting



- Useful to track down when you read uninitialized or undefined memory
 - Before it has been initialized
 - After it has been freed
- Memory Painting allows you to inject known values into memory on memory allocation or on memory free
 - Good values don't correspond to any valid address
 - Have a distinctive look
 - When cast to different types

Memory Hoarding



The screenshot shows a configuration window titled "Hoard deallocated memory". It contains several settings:

- A checked checkbox for "Hoard deallocated memory".
- A "Maximum KB to hoard:" label followed by a numeric input field set to "256".
- A "Maximum blocks to hoard:" label followed by a numeric input field set to "32".
- A checked checkbox for "Automatically release hoarded blocks when memory gets low".
- A "Warn & halt execution when hoard drops below threshold:" label followed by a dropdown menu set to "Off".
- A "Current Size:" label followed by two input fields: "0 KB" and "0 Blocks".

- **Memory Hoarding**
 - Stops the memory manager from reusing memory blocks
 - Can detect certain memory errors
- **Hoard Low Memory Controls**
 - Automatically release hoarded memory when available memory gets low, allowing your program to run longer
- **Hoard Low Memory events**
 - MemoryScape can stop execution as notification that the hoard dropped below a particular threshold. This provides an indication that the program is getting close to running out of memory.

Heap Graphical View

MemoryScape 3.2.3-0

File Tools Window Help

Home Memory Reports Manage Processes Memory Debugging Options Tips

Summary Leak Detection Heap Status Memory Usage Corrupted Memory Memory Comparisons

May 11, 2012

Save Data
Export Memory Data

Heap Status Reports
Source Report
Backtrace Report

Other Reports Category
Leak Detection Report
Memory Usage Report
Corrupted Memory Report
Compare Memory Usage

Other Tasks
Manage Filters

Process Selection

Process V

Parallel Job filter

MPI_COMM_WORLD

filterapp-mpi

filterapp-mpi

filterapp-mpi

filterapp-mpi

filterapp-mpi

Heap Status Graphical Report

Options

☐ Detect Leaks ☐ Relative to Baseline ☐ Enable Filtering

Leaked Block

Process 1: filterapp-mpi.1

0x0949d058 - 0x094d2c00 (214.91KB)

Heap Information Backtrace/Source Memory Content

Overall Totals

Category	Bytes
Heap	
<input type="checkbox"/> Allocated	81.55KB
<input type="checkbox"/> Deallocated	129.88KB
<input type="checkbox"/> Hoarded	0
<input type="checkbox"/> Leaked	Unknown
<input type="checkbox"/> Red Zones	0

Selected Block

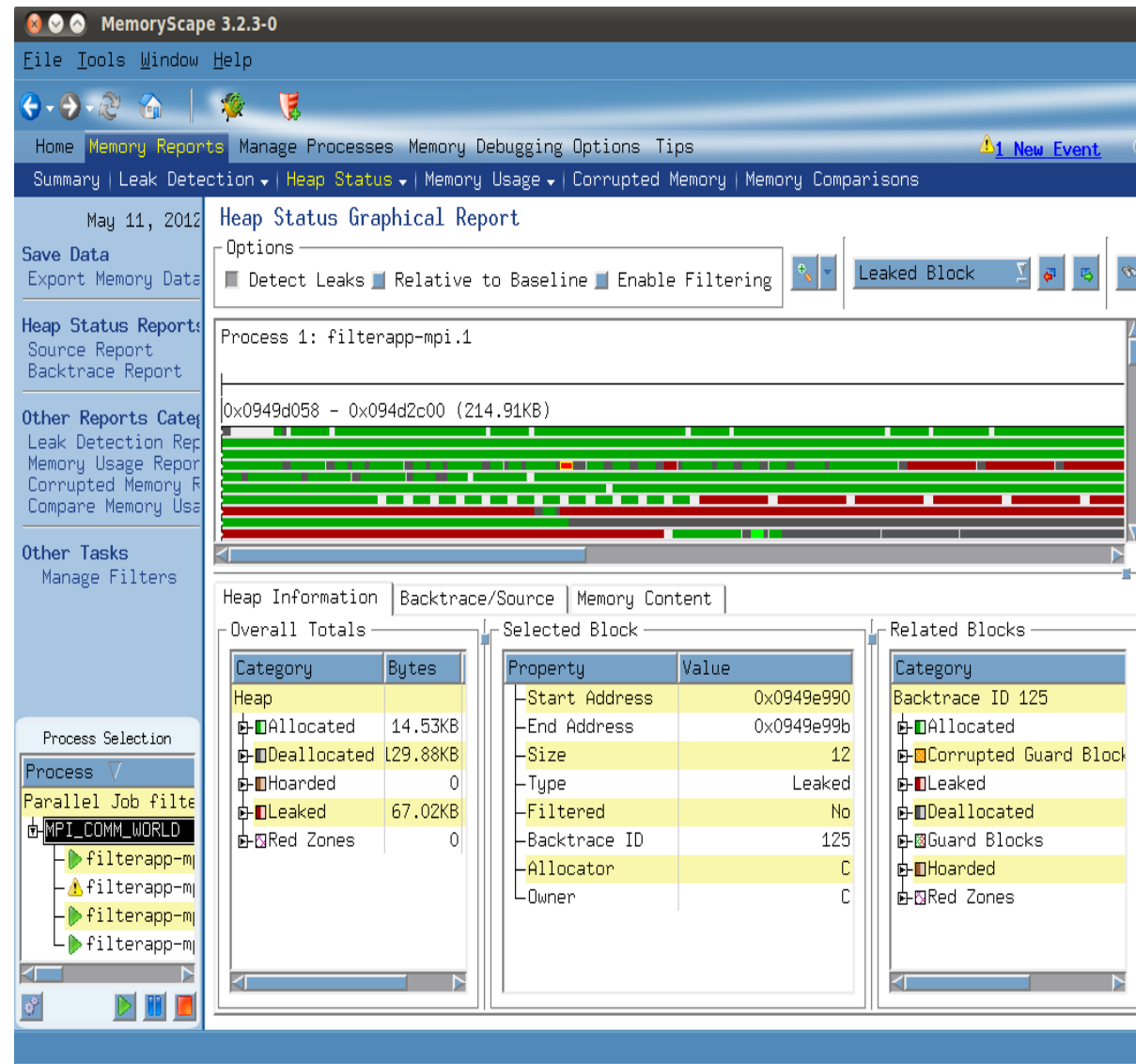
Property	Value
Start Address	0x0949d098
End Address	0x0949d0bb
Size	36
Type	Allocated
Filtered	No
Backtrace ID	3
Allocator	C
Owner	C

Related Blocks

Category
Backtrace ID 3
<input type="checkbox"/> Allocated
<input type="checkbox"/> Corrupted Guard Block
<input type="checkbox"/> Deallocated
<input type="checkbox"/> Guard Blocks
<input type="checkbox"/> Hoarded
<input type="checkbox"/> Leaked
<input type="checkbox"/> Red Zones

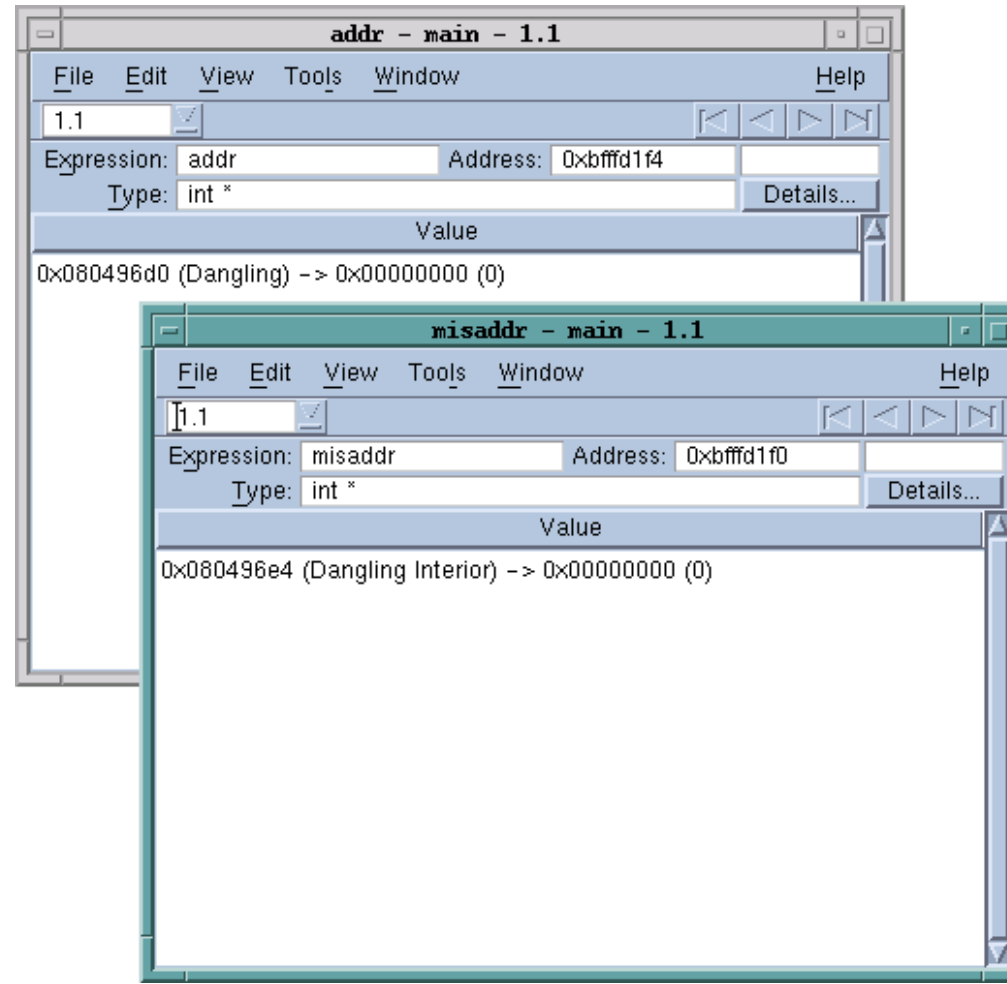
Leak Detection

- Leak Detection
 - Based on Conservative Garbage Collection
 - Can be performed at any point in runtime
 - Helps localize leaks in time
- Multiple Reports
 - Backtrace Report
 - Source Code Structure
 - Graphically Memory Location



Dangling Pointer Detection

Classic UI Only



Memory Comparisons

- “Diff” live processes
 - Compare processes across cluster
- Compare with baseline
 - See changes between point A and point B
- Compare with saved session
 - Provides memory usage change from last run

The screenshot displays the MemoryScape 3.2.3-0 application window. The main menu includes File, Tools, Window, and Help. The toolbar contains various icons for navigation and analysis. The 'Memory Reports' menu is open, showing options like Summary, Leak Detection, Heap Status, Memory Usage, Corrupted Memory, and Memory Comparisons. The 'Memory Comparisons' report is active, showing a comparison between Session 1 (filterapp-mpi.0) and Session 2 (filterapp-mpi.1). The report includes a table with columns for Process, Bytes Session 2, Bytes Session 1, Bytes Difference, and Count Session 2. The table lists several processes and their memory usage changes. A 'Process Selection' panel on the left shows a tree view of the process hierarchy, including MPI_COMM_WORLD and filterapp-m. The bottom of the window shows 'Session 1 Source' and 'Session 2 Source' tabs.

MemoryScape 3.2.3-0

File Tools Window Help

Home Memory Reports Manage Processes Memory Debugging Options Tips

Summary | Leak Detection | Heap Status | Memory Usage | Corrupted Memory | Memory Comparisons

May 11, 2012

Save View Options
Save Report...
Export Memory Data

Other Reports Category
Heap Status Report
Memory Usage Report
Corrupted Memory Report
Compare Memory Usage

Process Selection

Process
Parallel Job filter
MPI_COMM_WORLD
filterapp-m
filterapp-m
filterapp-m
filterapp-m

Memory Comparison Report

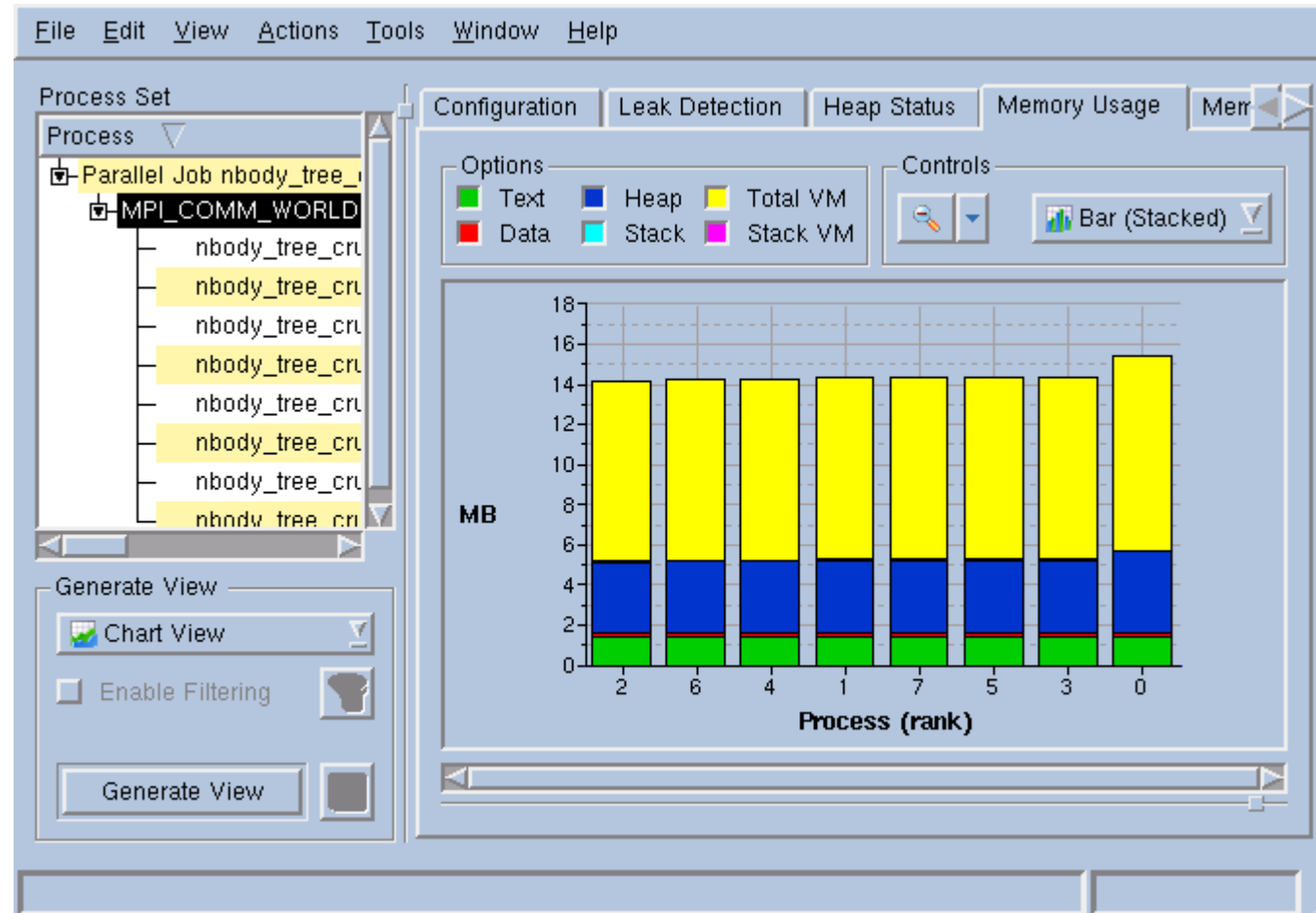
Data Source
Allocations
Leaks
Deallocations
Hoard
Red Zones

Process Comparisons
Session 1: filterapp-mpi.0
Session 2: filterapp-mpi.1
Reverse Diff

Process	Bytes Session 2	Bytes Session 1	Bytes Difference	Count Session 2
filterapp-mpi.0/filterap...	69.36KB	194	69.17KB	149
filterapp-mpi	69.36KB	194	69.17KB	149
myClassB.cxx	66.01KB	0	66.01KB	131
myClassA.cxx	2.00KB	0	2.00KB	4
new_allocator.h	1024	0	1024	1
main.cxx	192	0	192	7
stl_algobase.h	162	194	-32	6

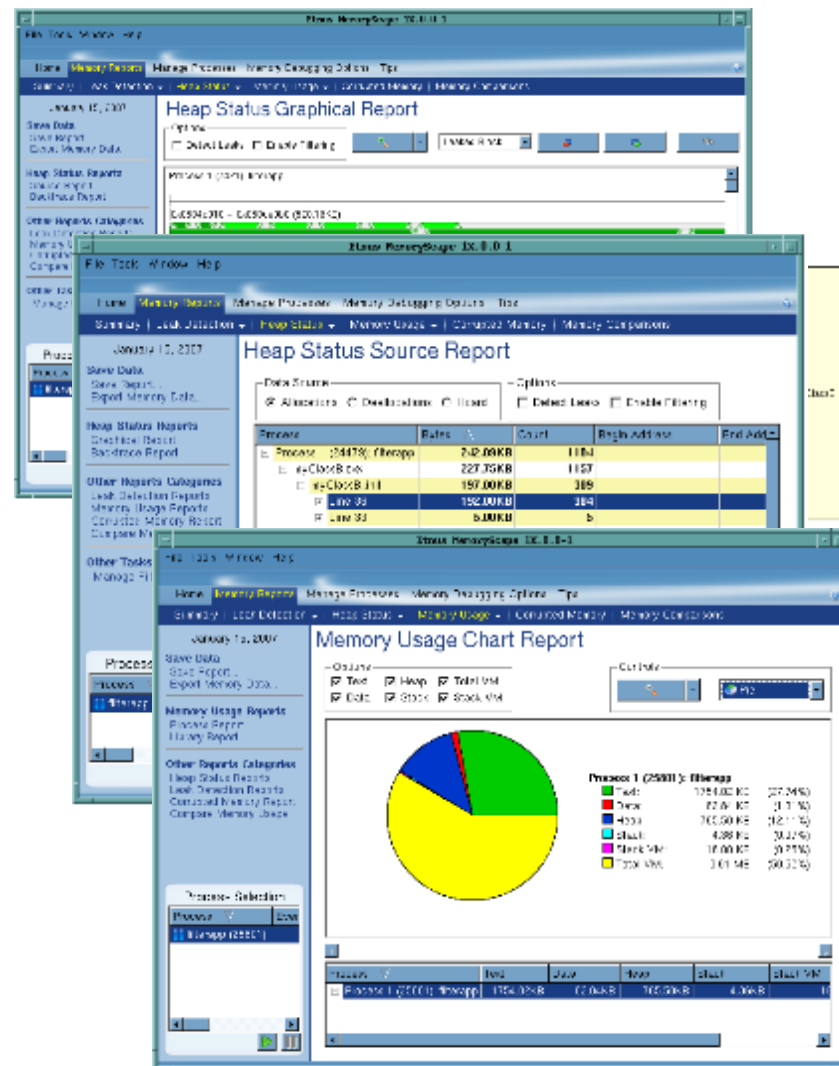
Session 1 Source Session 2 Source

Memory Usage Statistics



Memory Reports

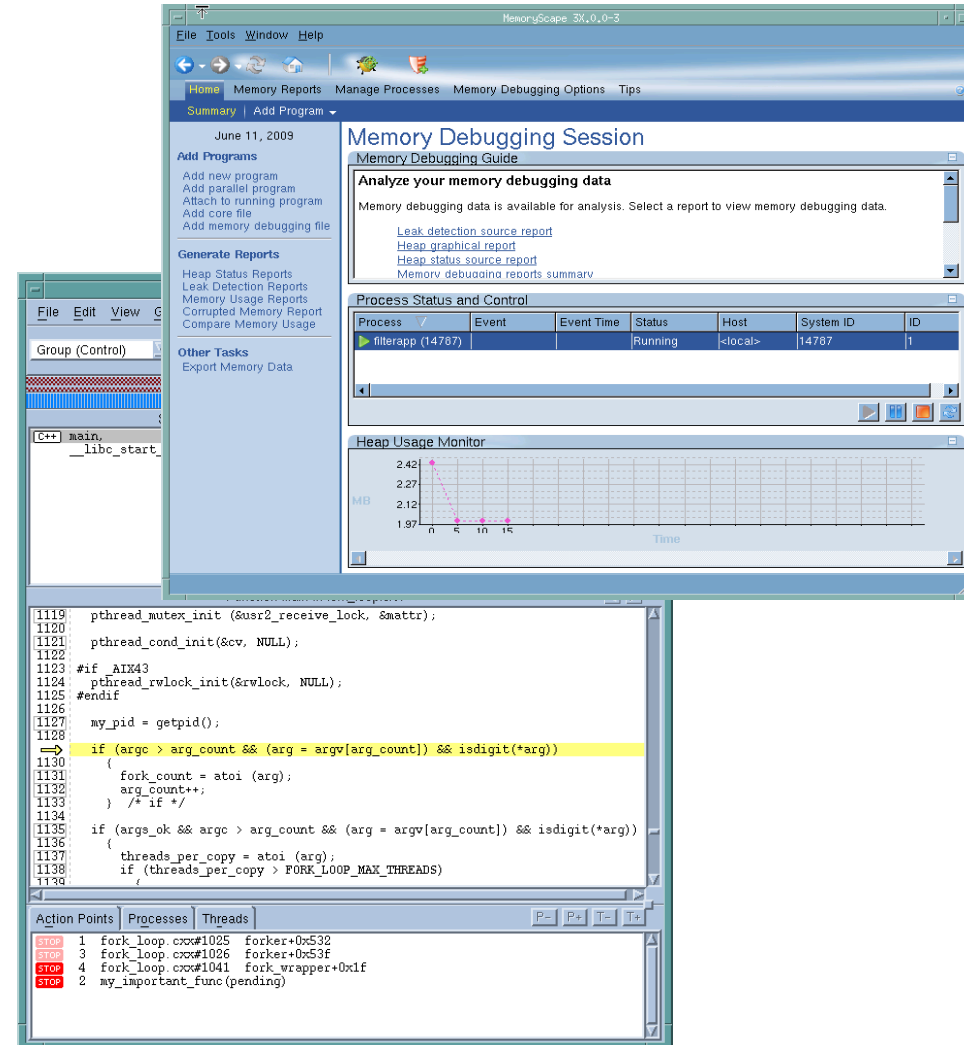
- Multiple Reports
 - Memory Statistics
 - Interactive Graphical Display
 - Source Code Display
 - Backtrace Display
- Allow the user to
 - Monitor Program Memory Usage
 - Discover Allocation Layout
 - Look for Inefficient Allocation
 - Look for Memory Leaks



TotalView and MemoryScape

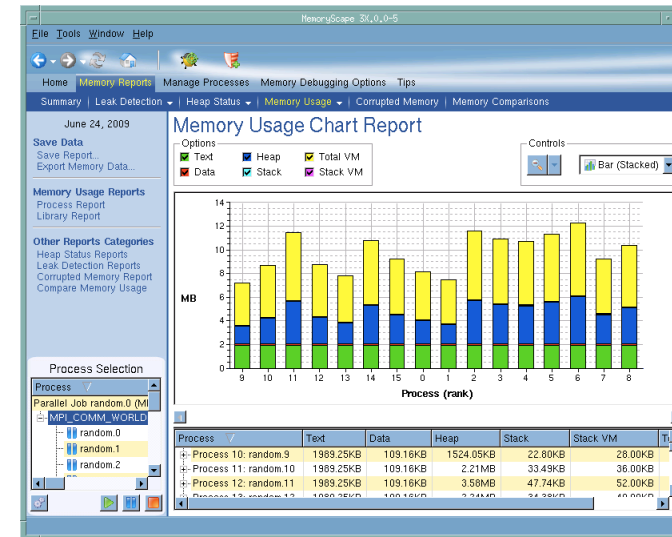
Classic UI Only

- You can use TotalView and MemoryScape together on an application.
- More precise control
 - Breakpoints
 - Stepping
- Visibility into
 - Memory behavior
 - Variable contents
- Advanced features
 - Heap baseline
 - Dangling pointer annotation
 - Memory painting



Strategies for Memory Debugging in Parallel

- Run the application and see if memory events are detected
- View memory usage across the MPI job
 - Compare memory footprint of the processes
 - Are there any outliers? Are they expected?
- Gather heap information in all processes of the MPI job
 - Select and examine individually
 - Look at the allocation pattern.
Does it make sense?
 - Look for leaks
 - Compare with the 'diff' mechanism
 - Are there any major differences?
Are they expected?



Demo

- MemoryScape Demo

C++View

- C++View is a simple way for you to define type transformations

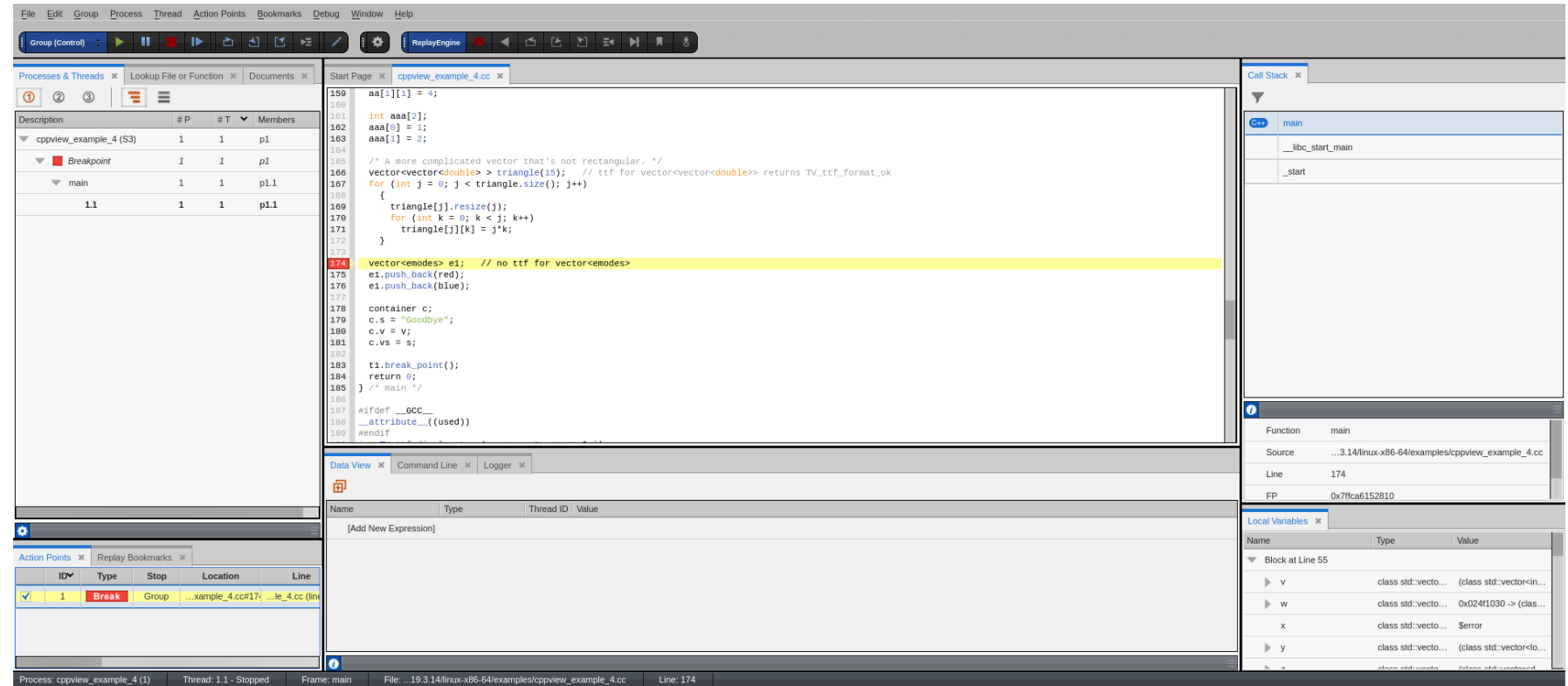
- Simplify complex data
- Aggregate and summarize
- Check validity

- Transforms

- Type-based
- Compose-able
- Automatically visible

- Code

- C++
- Easy to write
- Resides in target
- Only called by TotalView



C++ View API

```
#include "tv_data_display.h

int TV_ttf_display_type ( const T * );

int TV_ttf_add_row (
    const char *field_name,
    const char *type_name,
    const char *address );
```

C++ View Example

```
class A {
    int i;
    char *s;
};

class B {
    A a;
    double d;
};

int TV_ttf_display_type ( const A *a )
{
    (void) TV_ttf_add_row ( "i", TV_ttf_type_int, &(a->i) );
    (void) TV_ttf_add_row ( "s", TV_ttf_type_ascii_string, a->s );
}

int TV_ttf_display_type ( const B *b )
{
    (void) TV_ttf_add_row ( "a", "A", &(b->a) );
    (void) TV_ttf_add_row ( "d", "double", &(b->d) );
}
```

Type Transformation Facility (TTF)

TTF is a TotalView subsystem that

- Allows you to transform the way information appears
- Doesn't require changes to your code
- TTF transforms can be stored in a .tvd file for referencing during a debugging session or at TotalView startup

QT QString
class
transform

```
::TV::TTF::RTF::build_struct_transform {  
    name {^class QString$}  
    members {  
        { ascii { $wstring_u16 cast { * {d -> data} } } }  
    }  
}
```

Type Transformation Facility (TTF)

```
#include <stdio.h>
int main () {
    struct stuff {
        int month;
        int day;
        int year;
        char * pName;
        char * pStreet;
        char CityState[30];
    };
}
```

Type Transformation Facility (TTF)

```
::TV::TTF::RTF::build_struct_transform {  
    name {^struct stuff$}  
    members {  
        { year { year } }  
        { Name { * pName } }  
        { Street { * pStreet } }  
    }  
}
```



Name	Type	Threa	Value
▼ info	str...	1.1	(struct stuff)
Year	int	1.1	0x000007d4 (2004)
Name	\$st...	1.1	"John Smith"
Street	\$st...	1.1	"24 Prime Parkway, Su..."
[Add New Exp...]			

Stack Transformation Facility (STF)

- Hides stack frames
- Transforms stack frames
- Backbone for:
 - Python support
 - OpenMP support
- Useful for any glue code you want to hide
 - Language differences
 - Wrapper code

```
d1.<> dstacktransform list
Transformation Status: Enabled

Rules
  ID Transform      Operation  Filter
  1 RW_Python      modify    image('python[2-9]\.[0-9]+-dbg'),function('PyEval_EvalFrameEx')
  2 RW_Python      remove    image('python[2-9]\.[0-9]+-dbg')


Transforms
  Name      Implementation
  RW_Python  <built-in>
```



Stack Transformation Facility (STF)

```
Thread 1.1 has appeared
Created process 1 (11617), named "combined"
Thread 1.1 has appeared
Thread 1.1 has exited
Thread 1.1 hit breakpoint 8 at line 514 in "arrays(void)"
d1.<> dstacktransform enable
enabled
d1.<> dwhere
> 0 arrays      PC=0x00402c05, FP=0x7ffdf5d440 [/home/dstewart/Projects/TotalView/DemoDVD/src/combined.cxx#514]
  1 main        PC=0x004017db, FP=0x7ffdf5d4a0 [/home/dstewart/Projects/TotalView/DemoDVD/src/combined.cxx#33]
  2 __libc_start_main PC=0x7f42b17e2b13, FP=0x7ffdf5d560 [/lib64/libc.so.6]
  3 _start       PC=0x004016d4, FP=0x7ffdf5d568 [/home/dstewart/Projects/TotalView/DemoDVD/programs/combined]

d1.<> dstacktransform add -transform MY_TRANSFORMS -filter "function('^_start')" -operation remove
Rule 4 added.
4
d1.<> dwhere
> 0 arrays      PC=0x00402c05, FP=0x7ffdf5d440 [/home/dstewart/Projects/TotalView/DemoDVD/src/combined.cxx#514]
  1 main        PC=0x004017db, FP=0x7ffdf5d4a0 [/home/dstewart/Projects/TotalView/DemoDVD/src/combined.cxx#33]
  2 __libc_start_main PC=0x7f42b17e2b13, FP=0x7ffdf5d560 [/lib64/libc.so.6]

d1.<> |
```

Call Stack		Lookup File or Function
		
C++	arrays	
C++	main	
	__libc_start_main	
	_start	

Call Stack		Lookup File or Function
		
C++	arrays	
C++	main	
	__libc_start_main	

Demo

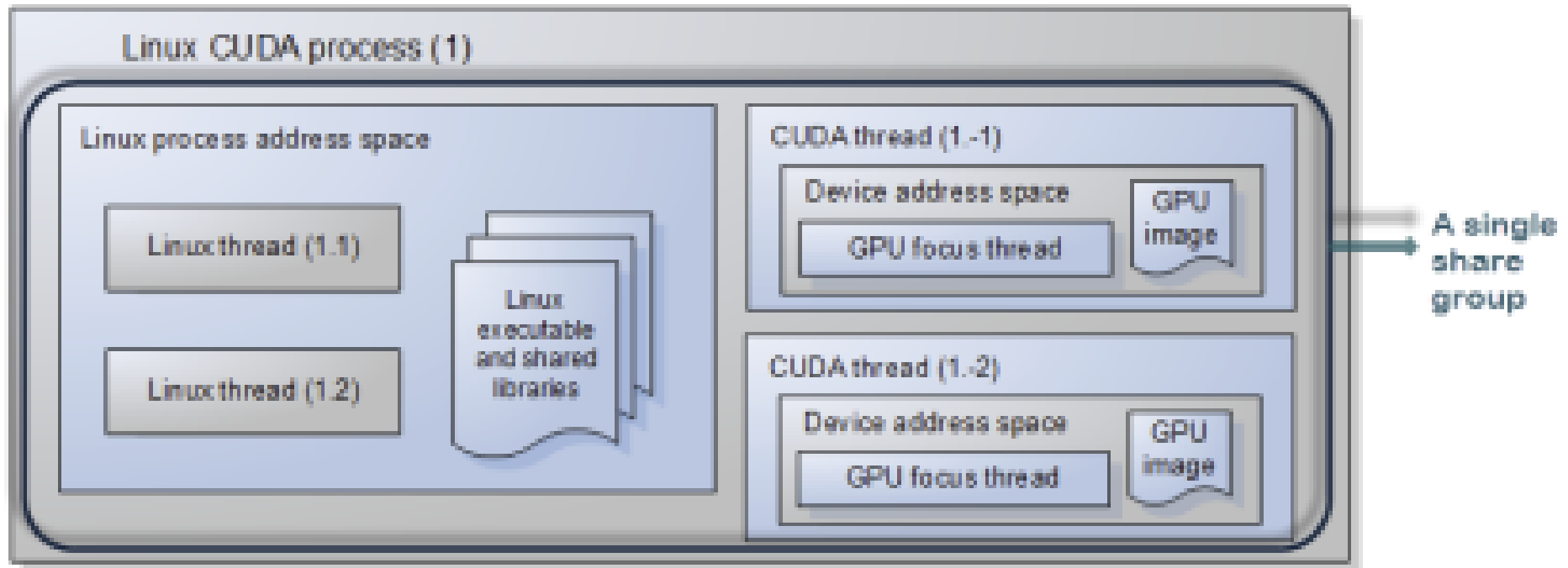
- TTF Demo (ttfdemo_customer example)

TotalView for CUDA

TotalView for the NVIDIA[®] GPU Accelerator

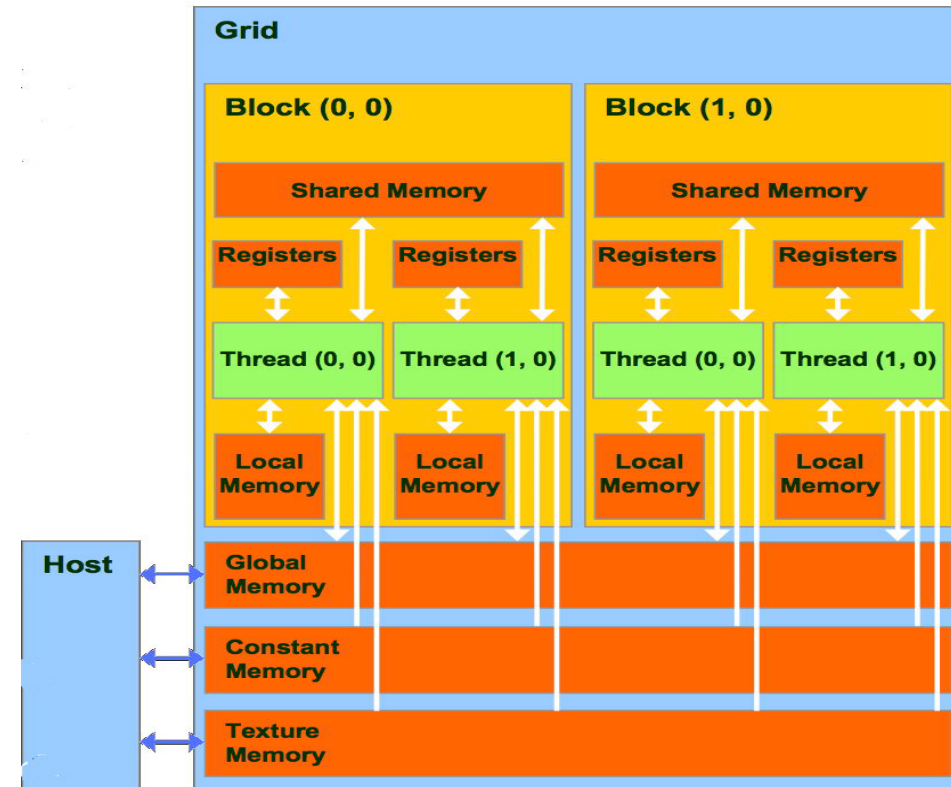
- NVIDIA Tesla, Fermi, Kepler, Pascal, Volta or Turing
- NVIDIA Ampere cards are in testing
- NVIDIA CUDA 9.2, 10.0 and 11.0
 - With support for Unified Memory
- Debugging 64-bit CUDA programs
- Features and capabilities include
 - Support for dynamic parallelism
 - Support for MPI based clusters and multi-card configurations
 - Flexible Display and Navigation on the CUDA device
 - Physical (device, SM, Warp, Lane)
 - Logical (Grid, Block) tuples
 - CUDA device window reveals what is running where
 - Support for types and separate memory address spaces
 - Leverages CUDA memcheck

TotalView CUDA Debugging Model



GPU Memory Hierarchy

- Hierarchical memory
 - Local (thread)
 - Local
 - Register
 - Shared (block)
 - Shared Memory
 - Global (GPU)
 - Global
 - Constant
 - Texture
 - System (host)



Supported Type Storage Qualifiers

@generic	An offset within generic storage
@frame	An offset within frame storage
@global	An offset within global storage
@local	An offset within local storage
@parameter	An offset within parameter storage
@iparam	Input parameter
@oparam	Output parameter
@shared	An offset within shared storage
@surface	An offset within surface storage
@texsampler	An offset within texture sampler storage
@texture	An offset within texture storage
@rtvar	Built-in runtime variables
@register	A PTX register name
@sregister	A PTX special register name

Control of Threads and Warps

- Warps advance synchronously
 - They share a PC
- Single step operation advances all GPU threads in the same warp
- Stepping over a `__syncthreads()` call will advance all relevant threads
- To advance more than one warp
 - Continue, possibly after setting a new breakpoint
 - Select a line and “Run To”

Compiling for CUDA debugging

When compiling an NVIDIA CUDA program for debugging, it is necessary to pass the `-g -G` options to the `nvcc` compiler driver. These options disable most compiler optimization and include symbolic debugging information in the driver executable file, making it possible to debug the application.

```
% /usr/local/bin/nvcc -g -G -c tx_cuda_matmul.cu -o tx_cuda_matmul.o
```

```
% /usr/local/bin/nvcc -g -G -Xlinker=-R/usr/local/cuda/lib64 \
tx_cuda_matmul.o -o tx_cuda_matmul
```

```
% ./tx_cuda_matmul
```

```
A:
```

```
[ 0][ 0] 0.000000
```

```
...output deleted for brevity...
```

```
[ 1][ 1] 131.000000
```

Compiling for debugging

Compiling for Fermi

`-gencode arch=compute_20,code=sm_20`

Compiling for Fermi and Tesla

`-gencode arch=compute_20,code=sm_20 -gencode arch=compute_10,code=sm_10`

Compiling for Kepler

`-gencode arch=compute_35,code=sm_35`

Compiling for Pascal

`-gencode arch=compute_60,code=sm_60`

Compiling for Volta

`-gencode arch=compute_70,code=sm_70`

A TotalView Session with CUDA

A standard TotalView installation supports debugging CUDA applications running on both the host and GPU processors.

TotalView dynamically detects a CUDA install on your system. To start the TotalView GUI or CLI, provide the name of your CUDA host executable to the `totalview` or `totalviewcli` command.

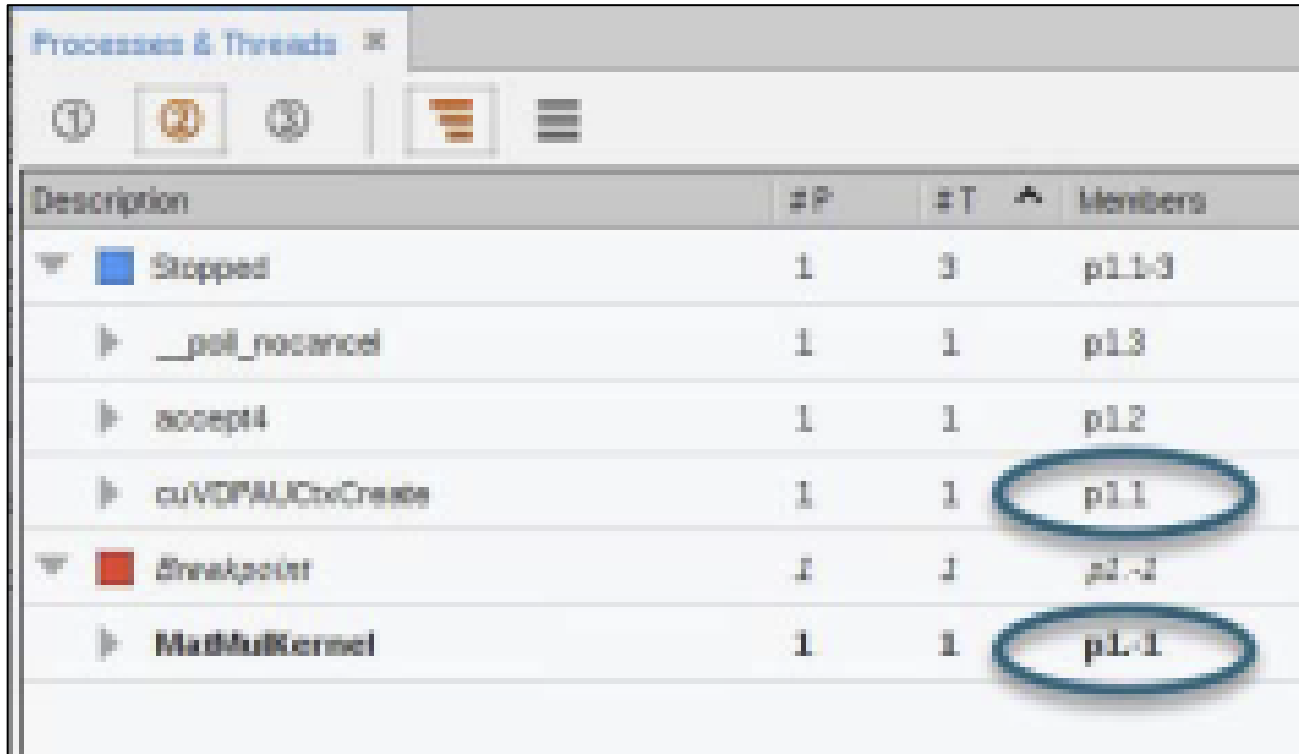
For example, to start the TotalView GUI on the sample program, use the following command:

```
% totalview tx_cuda_matmul
```

Source View Opened on CUDA host code

```
Start Page * tx_cuda_matmul.cu *  
144     for (int row = 0; row < height_; row++)  
145         for (int col = 0; col < width_; col++)  
146             A.elements[row * width_ + col] = row * 10.0 + col;  
147     return A;  
148 }  
149  
150 static void  
151 print_Matrix (Matrix A, const char *name)  
152 {  
153     printf("%s:\n", name);  
154     for (int row = 0; row < A.height; row++)  
155         for (int col = 0; col < A.width; col++)  
156             printf ("%s[%d][%d] %f\n", row, col, A.elements[row * A.stride + col]);  
157 }  
158  
159 // multiply an m*n matrix with an n*p matrix results in an m*p matrix.  
160 // Usage: tx_cuda_matmul [ m [ n [ p ] ] ]  
161 // m, n, and p default to 1, and are multiplied by BLOCK_SIZE.  
162 int main(int argc, char **argv)  
163 {  
164     // cudaSetDevice(0);  
165     const int m = BLOCK_SIZE * (argc > 1 ? atoi(argv[1]) : 1);  
166     const int n = BLOCK_SIZE * (argc > 2 ? atoi(argv[2]) : 1);  
167     const int p = BLOCK_SIZE * (argc > 3 ? atoi(argv[3]) : 1);  
168     Matrix A = cons_Matrix(m, n);  
169     Matrix B = cons_Matrix(n, p);  
170     Matrix C = cons_Matrix(m, p);  
171     MatMul(A, B, C);  
172     print_Matrix(A, "A");  
173     print_Matrix(B, "B");  
174     print_Matrix(C, "C");  
175     return 0;  
176 }
```

CUDA thread IDs and Coordinate Spaces



Description	#P	#T	Members
Stopped	1	3	p1.1-3
__poll_nocancel	1	1	p1.3
accept4	1	1	p1.2
cuVIDPAUCtxCreate	1	1	p1.1
Breakpoint	1	1	p1.-1
MathMulKernel	1	1	p1.-1

Host thread IDs have a positive thread ID (p1.1)

CUDA thread IDs have a negative thread ID (p1.-1)

GPU Physical and Logical Toolbars



Logical toolbar displays the Block and Thread coordinates.

Physical toolbar displays the Device number, Streaming Multiprocessor, Warp and Lane.

To view a CUDA host thread, select a thread with a positive thread ID in the Process and Threads view.

To view a CUDA GPU thread, select a thread with a negative thread ID, then use the GPU thread selector on the logical toolbar to focus on a specific GPU thread.

Displaying CUDA Program Elements

@local type qualifier indicates that variable A is in local storage.

"elements" is a pointer to a float in @generic storage

Pointer value 0x10216200000 is an offset within @generic storage.

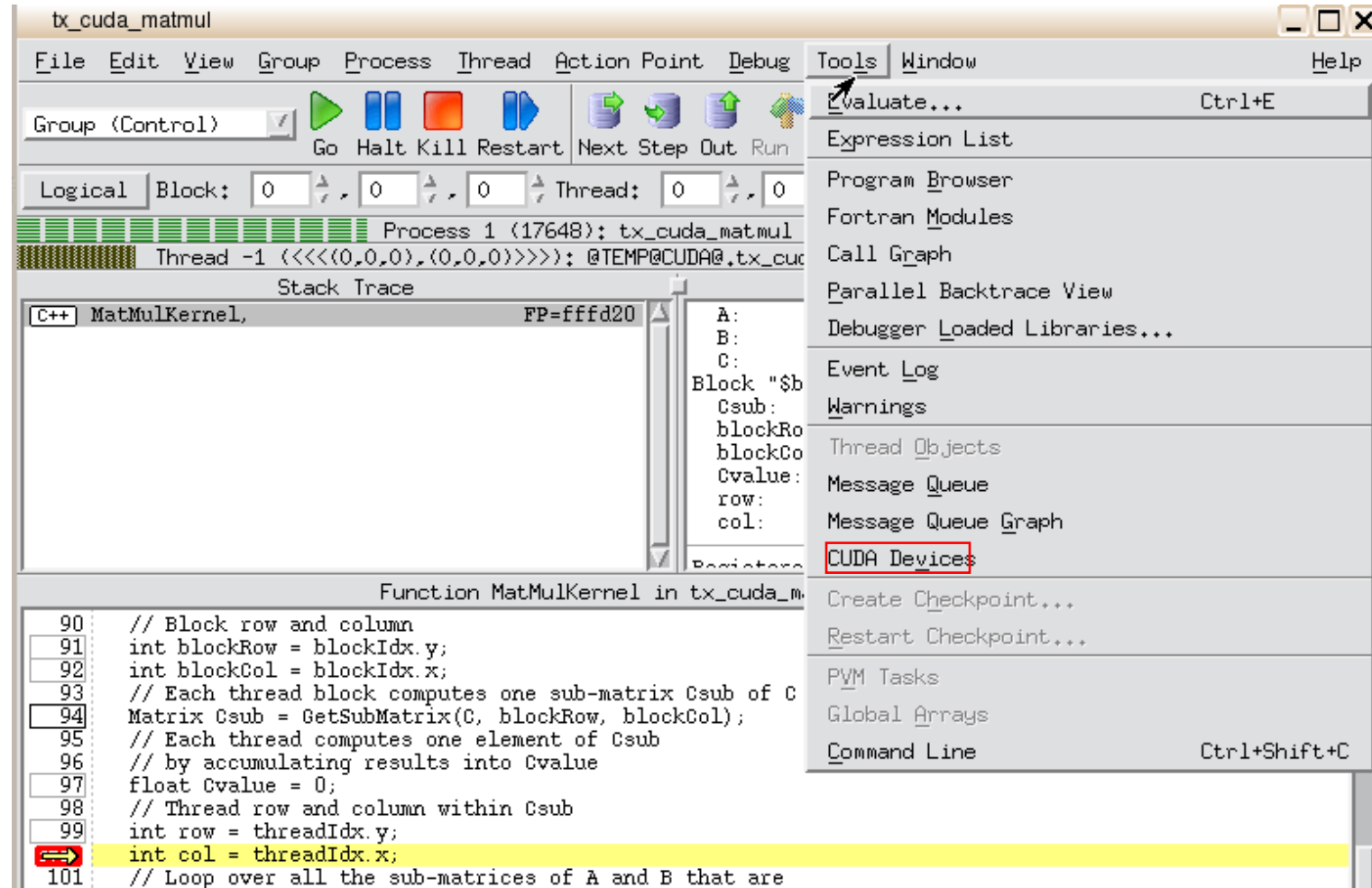
Name	Type	Thread ID	Value
A	Matrix @local	1-1	(Matrix @local)
width	int	1-1	0x00000002 (2)
height	int	1-1	0x00000002 (2)
stride	int	1-1	0x00000002 (2)
elements	float @generic *	1-1	0x10216200000 -> 0
*(elements)	@generic float	1-1	0
[Add New Expression]			

- The identifier @local is a TotalView built-in type storage qualifier that tells the debugger the storage kind of "A" is local storage.
- The debugger uses the storage qualifier to determine how to locate A in device memory

The CUDA Devices Window

Classic UI Only

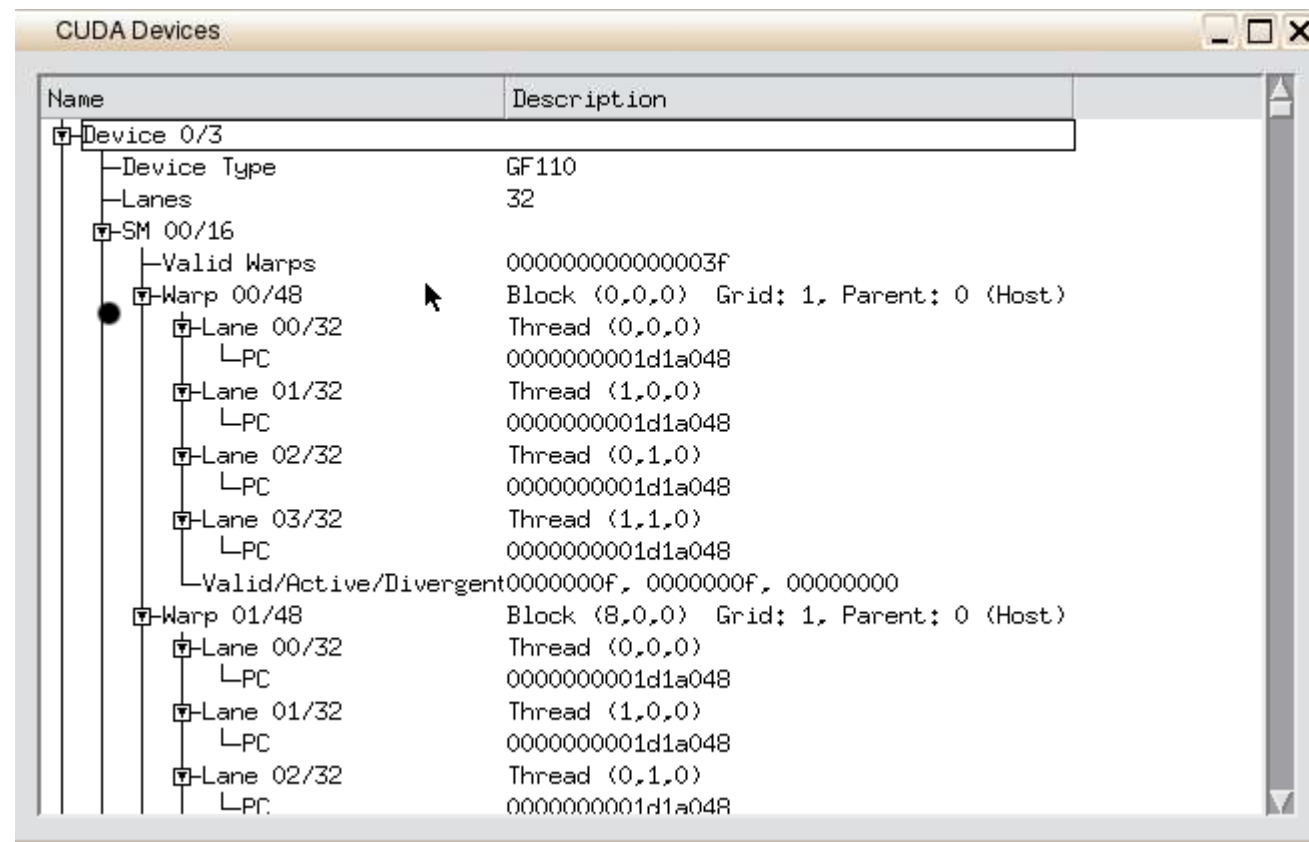
We can get an idea of what physical cards are accessible and what threads we can access with the CUDA devices window



The CUDA Devices Window

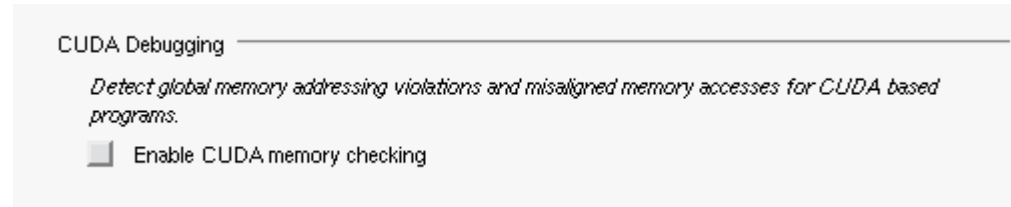
- The devices window shows us the type of device, the number of lanes available and the active SM's, warps and lanes

Classic UI Only



CUDA Memory Checker Feature

Classic UI Only



To enable CUDA memory checking in the new UI :

- Pass the `-cuda_memcheck` option to the `totalview` command, for example:
`totalview -cuda_memcheck`
- Set the `TV::cuda_memcheck` CLI state variable to `true`. For example:
`dset TV::cuda_memcheck true`

Demo

The screenshot displays the TotalView for HPC 2019X interface, titled "tx_cuda_matmul - Process 1, No current thread - TotalView for HPC 2019X (on microway2.totalviewtech.com)". The interface is divided into several panels:

- Top Panel:** Contains menu items (File, Edit, Group, Process, Thread, Action Points, Bookmarks, Debug, Window, Help) and a toolbar with various execution and debugging controls.
- GPU (Logical) Panel:** Shows GPU configuration details including Block, Thread, and GPU (Physical) settings.
- Processes & Threads Panel:** Displays a table of processes and threads. The table has columns for Description, #P, #T, and Members. The process "tx_cuda_matmul (S3)" is listed with 1 process and 1 thread, labeled "p1". A "Nonexistent" entry is also shown.
- Code Editor:** Shows the source code for "tx_cuda_matmul.cu". The code includes a loop for matrix multiplication and a main function that sets up the GPU and prints the results. The code is as follows:

```
154 for (int row = 0; row < A.height; row++)
155     for (int col = 0; col < A.width; col++)
156         printf ("%s%d][%s%d] %f\n", row, col, A.elements[row * A.stride + col]);
157 }
158
159 // multiply an m*n matrix with an n*p matrix results in an m*p matrix.
160 // usage: tx_cuda_matmul [ m [ n [ p ] ] ]
161 // m, n, and p default to 1, and are multiplied by BLOCK_SIZE.
162 int main(int argc, char **argv)
163 {
164     cudaSetDevice(0);
165     const int m = BLOCK_SIZE * (argc > 1 ? atoi(argv[1]) : 1);
166     const int n = BLOCK_SIZE * (argc > 2 ? atoi(argv[2]) : 1);
167     const int p = BLOCK_SIZE * (argc > 3 ? atoi(argv[3]) : 1);
168     Matrix A = cons_Matrix(m, n);
169     Matrix B = cons_Matrix(n, p);
170     Matrix C = cons_Matrix(m, p);
171     MatMul(A, B, C);
172     print_Matrix(A, "A");
173     print_Matrix(B, "B");
174     print_Matrix(C, "C");
175     return 0;
176 }
177
```
- Call Stack Panel:** Shows the current call stack, which is empty, indicating "No current thread".
- Local Variables Panel:** Shows the local variables, which are currently empty.
- Action Points Panel:** Displays a table of action points with columns for ID, Type, Stop, and Location.
- Data View Panel:** Shows a table of data with columns for Name, Type, Thread ID, and Value. The table is currently empty, with a placeholder "[Add New Expression]".

The status bar at the bottom indicates "Process: tx_cuda_matmul (1)" and "No current thread".

Debugging with TotalView's CLI

TotalView's CLI

- TotalView's CLI is a TCL based command line interface providing full access to the debugger
- Accessible through:
 - *Tools / Command Line* menu in Classic UI
 - Command Line view in new UI
 - Directly by running `totalviewcli`
- Using the CLI in the TotalView Reference Guide provides comprehensive documentation on all the CLI commands and options
 - http://docs.roguewave.com/totalview/current/html/index.html#page/Reference_Guide%2Ftotalviewref-part1.html

Using the CLI to examine data

`dgo`

Resumes execution of target process

`dhalt`

Suspends execution of target process

`dhelp`

Lists the available commands

`dlist`

Displays code relevant to current location

`dnext`

Steps source lines, stepping over subroutines

Using the CLI to examine data (2)

`dprint`

Prints the value of a variable or an expression

`dstatus`

Displays an aggregated view of the current processes and threads

`dstep`

Steps lines, stepping into subroutines

`dwhat`

Displays information about a symbol

`dwhere`

Displays locations in the call stack

Demo

- CLI Demo (combined example)

Batch Debugging with TVScript

tvscript

- A straightforward language for unattended and/or batch debugging with TotalView and/or MemoryScape
- Usable whenever jobs need to be submitted or batched
- Can be used for automation
- A more powerful version of printf, no recompilation necessary between runs
- Schedule automated debug runs with *cron* jobs
- Expand its capabilities using TCL

tvscript

```
tvscript [options] [filename] [ -a program_args]
```

options

TotalView and tvscript command-line options.

filename

The program being debugged.

-a program_args

Program arguments.

tvscript

- All of the following information is provided by default for each print
 - Process id
 - Thread id
 - Rank
 - Timestamp
 - Event/Action description
- A single output file is written containing all of the information regardless of the number of processes/threads being debugged

Supported tvscript events

Event Type	Event	Definition
General event	any_event	A generated event occurred.
Memory debugging event	addr_not_at_start	Program attempted to free a block using an incorrect address.
	alloc_not_in_heap	The memory allocator returned a block not in the heap; the heap may be corrupt.
	alloc_null	An allocation either failed or returned NULL; this usually means that the system is out of memory.
	alloc_returned_bad_alignment	The memory allocator returned a misaligned block; the heap may be corrupt.
	any_memory_event	A memory event occurred.
	bad_alignment_argument	Program supplied an invalid alignment argument to the heap manager.
	double_alloc	The memory allocator returned a block currently being used; the heap may be corrupt.
	double_dealloc	Program attempted to free an already freed block.
	free_not_allocated	Program attempted to free an address that is not in the heap.
	guard_corruption	Program overwrote the guard areas around a block.

Supported tvscript events

Event Type	Event	Definition
Source code debugging event	hoard_low_memory_threshold	Hoard low memory threshold crossed.
	realloc_not_allocated	Program attempted to reallocate an address that is not in the heap.
	rz_overrun	Program attempted to access memory beyond the end of an allocated block.
	rz_underrun	Program attempted to access memory before the start of an allocated block.
	rz_use_after_free	Program attempted to access a block of memory after it has been deallocated.
	rz_use_after_free_overrun	Program attempted to access memory beyond the end of a deallocated block.
	rz_use_after_free_underrun	Program attempted to access memory before the start of a deallocated block.
	termination_notification	The target is terminating.
	actionpoint	A thread hit an action point.
	error	An error occurred.
Reverse debugging	stopped_at_end	The program is stopped at the end of execution and is about to exit.

Supported tvscript actions

Action Type	Action	Definition
Memory debugging actions	check_guard_blocks	Checks all guard blocks and write violations into the log file.
	list_allocations	Writes a list of all memory allocations into the log file.
	list_leaks	Writes a list of all memory leaks into the log file.
	save_html_heap_status_source_view	Generates and saves an HTML version of the Heap Status Source View Report.
	save_memory_debugging_file	Generates and saves a memory debugging file.
	save_text_heap_status_source_view	Generates and saves a text version of the Heap Status Source View Report.
Source code debugging actions	display_backtrace [-level <i>level-num</i>] [<i>num_levels</i>] [<i>options</i>]	<p>Writes the current stack backtrace into the log file.</p> <p>-level <i>level-num</i> sets the level at which information starts being logged.</p> <p><i>num_levels</i> restricts output to this number of levels in the call stack.</p> <p>If you do not set a level, tvscript displays all levels in the call stack.</p> <p><i>options</i> is one or more of the following:</p> <ul style="list-style-type: none">-[no]show_arguments-[no]show_fp-[no]show_fp_registers-[no]show_image-[no]show_locals-[no]show_pc-[no]show_registers

Supported tvscript actions

Action Type	Action	Definition
	print [-slice { <i>slice_exp</i> } { <i>variable</i> <i>exp</i> }	Writes the value of a variable or an expression into the log file. If the variable is an array, the -slice option limits the amount of data defined by <i>slice_exp</i> . A slice expression is a way to define the slice, such as var[100:130] in C and C++. (This displays all values from var[100] to var[130] .) To display every fourth value, add an additional argument; for example, var[100:130:4] . For additional information, see “Examining Arrays” in the <i>TotalView for HPC User Guide</i> .
Reverse debugging actions	enable_reverse_debugging	Turns on ReplayEngine reverse debugging and begins recording the execution of the program.
	save_replay_recording_file	Saves a ReplayEngine recording file. The file name is of the form <ProcessName>-<PID>_<DATE>.<INDEX>.recording.

tvscript examples

Simple example

```
tvscript \  
-create_actionpoint "method1=>display_backtrace -show_arguments" \  
-create_actionpoint "method2#37=>display_backtrace \  
    -show_locals -level 1" \  
-event_action "error=>display_backtrace -show_arguments \  
    -show_locals" \  
-display_specifiers "noshow_pid,noshow_tid" \  
-maxruntime "00:00:30" \  
~/work/filterapp /filterapp -a 20
```

MPI example

```
tvscript -mpi "Open MPI" -tasks 4 \  
-create_actionpoint \  
"hello.c#14=>display_backtrace" \  
~/tests/MPI_hello
```

tvscript examples

Memory Debugging example

```
tvscript -maxruntime "00:00:30" \  
-event_action "any_event=save_memory_debugging_file" \  
-guard_blocks -hoard_freed_memory -detect_leaks \  
~/work/filterapp -a 20
```

ReplayEngine example

```
tvscript \  
-create_actionpoint "main=>enable_reverse_debugging" \  
-event_action "stopped_at_end=>save_replay_recording_file" \  
filterapp
```

Demo

- TVScript demo (tvscript --script_file file tvscript_example.tvd ex2)

TotalView Roadmap

Recent TotalView Release Features (2020)

- **2020.3 (Nov 2020)**

- TotalView Remote UI
- OpenMP Debug API v5.0 support
- TotalView Student license changes
- GCC 10, Fedora 32, Ubuntu 20.04

- **2020.2 (Sep 2020)**

- CUDA 11 support
- Remote Display Client updated
- TotalView Solaris platform ported to 64 bits
- “Dive-in-all” to view a structure member across an array
- Easily focus on specific data in new data views

- **2020.1 (May 2020)**

- TotalView FlexNet Publisher License update
- macOS Catalina support
- TotalView stepping and local variable view performance updates
- Array statistics view auto update
- FlexNet Embedded failover server support

- **2020 (Feb 2020)**

- Display thread names and lightweight process IDs
- Array statistics view
- Rebranding to use “TotalView by Perforce” logo

Common TotalView Usage Hints

Common TotalView Usage Hints

- TotalView can't find the program source
 - Did you compile with -g ?
 - How to adjust the TotalView search paths? Preferences -> Search Path
- Python Debugging
 - Making sure proper system debug packages are installed for Python
- X11 forwarding performance
 - If users are forwarding X11 displays through ssh TotalView UI performance can be bad
- Understanding different ways to stop program execution with TotalView Action Points
 - Using a watchpoint on a local variable
- Focus
 - Diving on a variable that is no longer in scope. Check the Local Variables window for in scope variables
 - Totalview doesn't change focus to the thread hitting a breakpoint. Set Action Point Preferences to "Automatically focus on threads/processes at breakpoint"

Common TotalView Usage Hints (cont.)

- MPI Debugging
 - Differences in launching MPI job from within the TotalView UI vs the command line.
 - TotalView runs an MPI program without stopping. Set the Parallel Preferences to “Ask What To Do” in After Attach Behavior
 - Using wrong attributes in processes and threads view
- Reverse Debugging
 - Running out of memory by not setting the maximum memory allocated to ReplayEngine
 - Defer turning on reverse debugging until later in program execution to avoid slow initialization phases
 - Adjust reverse debugging circular buffer size to reduce resources
- Memory Debugging
 - Starting with All memory debugging options enabled rather than Low
 - Not setting a size restriction for Red Zones
 - Issues with getting memory debugging turned on in an MPI job. May have to set LD_PRELOAD environment variable or worst case, prelink HIA

Common TotalView Usage Hints (cont.)

- Differences in functionality between new UI and classic UI
 - How to switch between them. Preferences -> Display or `totalview -newUI` and `totalview -oldUI`
 - Where the gaps still are in functionality
- Reverse Connect with `tvconnect`
 - When I use Reverse Connect I get the following obscure message: *myProgram is an invalid or incompatible executable file format for the target platform*
 - The message indicates an incompatible file format but most often this occurs if the program provided to `tvconnect` for TotalView to debug cannot be found. The easiest way to resolve problem is to provide the full path to the target application, e.g., `tvconnect /home/usr/myProgram`
- How do I get help?
 - How to submit a support ticket? `techsupport@roguewave.com`
 - Where is TV documentation (locally and on the internet). <https://help.totalview.io/>
 - Are there videos I can watch to learn how to use TotalView? <https://totalview.io/support/video-tutorials>

LBL/NERSC - TotalView Usage Hints

LBL/NERSC TotalView Usage Hints

- Remote UI
 - Configure cori.nersc.gov Remote UI:
 - Connection Name: cori.nersc.gov
 - Remote Host(s): username@cori.nersc.gov
 - TotalView Remote Installation Directory: /usr/common/software/toolworks/totalview.default/bin
 - For now, load TotalView modules needed in your .bash_profile.ext
 - module load totalview
- Use Reverse Connect (tvconnect) to easily connect back and launch applications/jobs
 - tvconnect stays active until user detaches from the UI or ctrl-c tvconnect once done
 - Batch Submission with sbatch:
 - ```
#!/bin/bash
#SBATCH --qos=debug
#SBATCH --time=5
#SBATCH --nodes=1
#SBATCH --tasks-per-node=4
#SBATCH --constraint=haswell
module load openmpi
tvconnect srun primeCount
```
  - Pre-allocated node – make sure to use full paths
    - **tvconnect** `which srun` -n 4 `pwd`/primeCount

Q&A

# Contact us

- Bill Burns (Senior Director of Product Development and Product Manager)

[bburns@perforce.com](mailto:bburns@perforce.com)

- Dean Stewart (Senior Sales Engineer)

[dstewart@perforce.com](mailto:dstewart@perforce.com)